Biblioteczka Komputer Świat



DVD I KŚ+
- PLIKI SZKOLENIOWE
- NARZĘDZIA DLA PROGRAMISTÓW

ISO płyty i skrypty do pobrania z ksplus.pl

ZACZNIJ PROGRAMOWAĆ

# python W PIGUŁCE



KURS DLA POCZĄTKUJĄCYCH I ZAAWANSOWANYCH PROGRAMISTÓW

## KROK PO KROKU OPANUJESZ:

- instrukcje warunkowe
- pętle
- definiowanie funkcji
- stosowanie klas
- obsługę błędów
- programowanie obiektowe
- użycie Pythona do komunikacji z internetem




# Z TĄ KSIĄŻKĄ E-WYDANIE GRATIS

**Poniżej znajduje się płyta z kodem bonusowym dającym dostęp do e-wydania tej książki w serwisie KŚ+ (ksplus.pl) oraz pliku ISO z cyfrową wersją płyty do pobrania.**

## NA PŁYCIE DVD

**Płyta dołączona do tej książki zawiera zestaw najlepszych i najpopularniejszych darmowych narzędzi do programowania w języku Python. Na DVD znajdują się zintegrowane środowiska programistyczne (IDE), edytory kodu źródłowego oraz pliki szkoleniowe do zadań opisanych w książce.**

---

**Jeżeli brakuje płyty, poinformuj sprzedawcę lub redakcję: pomoc@komputerswiat.pl**

python

PŁYTA JEST DODATKIEM DO KSIĄŻKI
KOMPUTER ŚWIAT BIBLIOTECZKA
4|2021

- środowiska programistyczne
- edytory kodu źródłowego
- materiały szkoleniowe

Komputer Świat Biblioteczka

KOD DO E-WYDANIA I BONUSÓW W KSPLUS.PL

**Kod bonusowy należy zarejestrować w KŚ+ (ksplus.pl)**

Biblioteczka Komputer Świat

KRZYSZTOF DZIEDZIC

# python W PIGUŁCE

ringier axel springer

**AUTOR:** Krzysztof Dziedzic

**REDAKTORZY PROWADZĄCY:** Rafał Kamiński, Agnieszka Al-Jawahiri

**PRZYGOTOWANIE PŁYTY:** Mariusz Michalski

**PROJEKT OKŁADKI:** Robert Dobrzyński

**SKŁAD I ŁAMANIE:** Mariusz Rybak

**KOREKTA:** Jolanta Rososińska

**WYDAWCA:**

RINGIER AXEL SPRINGER POLSKA Sp. z o.o.
02-672 Warszawa, ul. Domaniewska 49
tel. 12 2600200 (BOK)
www.ringieraxelspringer.pl

ISBN 978-83-8250-084-4



Warszawa 2021

**BUSINESS PROJECT MANAGER:** Paweł Bulwan

**DRUK I OPRAWA:**
Drukarnia im. Adama Półtawskiego, Kielce

**EGZEMPLARZE ARCHIWALNE:**
literia.pl, prenumerata.axel@qg.com

**E-WYDANIA, E-PRENUMERATA:**
ksplus.pl

**KONTAKT:**
redakcja@komputerswiat.pl

**INTERNET:**
komputerswiat.pl, ksplus.pl

Płyta DVD jest dodatkiem do książki

ringier
axel springer

# 1 Przygotowanie do pracy z Pythonem

**W tym kursie programowania poznamy jeden z najbardziej popularnych języków programowania ostatnich lat. Za pomocą Pythona są tworzone aplikacje sieciowe, jest on też wykorzystywany w skomplikowanych obliczeniach, można nawet tworzyć w nim gry. Zanim zaczniemy uczyć się korzystać z Pythona, musimy poznać kilka podstawowych informacji o nim i przygotować środowisko pracy**

Warto wiedzieć, że Python jest stosowany przez takie firmy, jak Facebook, Google, Dropbox czy Netflix.

Każda nowoczesna aplikacja sieciowa może zostać przynajmniej częściowo napisana lub uzupełniona w języku Python, nawet wtedy, gdy została już oddana do użytku. Python pozwala rozbudowywać gotowe programy, także napisane w innych językach, na przykład w Javie lub C++, dzięki temu, że można wykorzystywać pojedyncze moduły tworzone właśnie w Pythonie. Jeśli więc mamy jakieś nowe zadanie do zrealizowania w ramach już istniejącego programu – potrzebujemy dodać skrypt czy większą część aplikacji, która ma wykonywać konkretne działania – zawsze możemy skorzystać z Pythona.

Python jest językiem wysokiego poziomu. Oznacza to, że jego składnia oraz słowa kluczowe ułatwiają rozumienie kodu przez programistę. Dzięki temu pisanie w nim jest dość naturalne. Zanim jednak napisany kawałek kodu będzie mógł zostać uruchomiony, musi być zinterpretowany, czyli – w skrócie – przekształcony z języka zrozumiałego dla człowieka na język zrozumiały dla komputera.

Python powstał w 1991 roku i od tamtego czasu przeszedł wiele mniejszych i większych rewolucji. Obecnie praktycznie wszyscy zaczynający programowanie w tym języku uczą się składni i obsługi wersji 3.x. Niestety, część napisanych w poprzednich latach aplikacji wykorzystuje wersję 2.x, która jest dość znacząco inna i wymaga innego typu bibliotek oraz programowania.

W tym kursie skupimy się na wersji 3.x, a w wypadkach, w których jest to istotne, będziemy też poznawać rozwiązania stosowane w starszej wersji.

```
int silnia(int x) {
    if (x == 0) return 1;
    else return x * silnia(x-1);
}
```

Przykład kodu silni bez wcięć napisanego w C

```
def silnia(x):
    if x == 0:
        return 1
    else:
        return x * silnia(x-1)
```

Przykład kodu silni z wcięciami napisanego w Pythonie

## Python a inne języki

Python charakteryzuje się tym, że stosowane w nim wartości mają typy, jest językiem z typami dynamicznymi (w przeciwieństwie do Javy).
Dodatkowo wszystko jest w nim obiektem, możliwe jest dziedziczenie z dowolnego typu, a nawet z liczb całkowitych.
Dużą różnicą między Pythonem a C++ i Javą jest brak w nim enkapsulacji (inaczej hermetyzacji), czyli ukrywania danych składowych lub metod obiektów, aby były dostępne tylko metodom wewnętrznym danej klasy.
A najbardziej rzucającą się w oczy charakterystyczną cechą Pythona jest struktura kodu z wcięciami. To niezwykle ważne – pisząc kod w Pythonie, zawsze musimy pamiętać, aby dzielić bloki kodu poprzez odpowiednio dopasowane wcięcia. Bardzo ułatwia to późniejsze czytanie i rozumienie kodu.

## Zalety oraz wady Pythona

Python jest językiem programowania o bardzo szerokim przeznaczeniu, który może być wykorzystywany do tworzenia skryptów lub całych aplikacji. Bardzo często można spotkać się z sytuacją, że zamiast terminu program używa się pojęcia skrypt w odniesieniu do kodów w języku Python. Jest on często określany jako zorientowany obiektowo skryptowy język programowania.

**Zalety języka Python:**

- **Wysoka jakość oprogramowania** – kod tworzony za pomocą Pythona jest bardzo czytelny, można do niego wracać i bez problemów go rozbudowywać.
Dzięki dużej spójności łatwo można zrozumieć kod pisany przez innych programistów.
A cechą programowania zorientowanego obiektowo (OOP) jest to, że są dostępne mechanizmy umożliwiające ponowne wykorzystywanie kodu w różnych skryptach.
- **Wydajność** – dzięki dobrze przemyślanej składni statystycznie kod napisany w Pythonie to zaledwie od jednej trzeciej do jednej piątej kodu pisanego w Javie lub C czy też C++. Mniejsza liczba znaków oznacza szybsze tworzenie skryptów.
Dodatkowo programy tworzone w Pythonie nie wymagają kompilacji, dzięki czemu mogą być uruchamiane natychmiast – co jest ogromną zaletą przy tworzeniu rozbudowanych aplikacji sieciowych.
- **Uniwersalność** – zdecydowana większość programów napisanych w Pythonie działa bez modyfikacji na wszystkich popularnych platformach. Oznacza to, że jeśli stworzymy skrypt w Windows, a chcemy z niego korzystać w innym systemie, na przykład w Linu-

## PYTHON 2.X A PYTHON 3.X

Bardzo wiele kodów zostało już napisanych w wersji Pythona 2.x i jest nadal wykorzystywanych w przeróżnych aplikacjach i narzędziach. Zaczynając naukę Pythona, oczywiście należy przede wszystkim poznawać wersję 3.x, trzeba też jednak orientować się, jakie są podstawowe różnice pomiędzy tymi wersjami. Uważa się, że Python 3.x jest językiem czystszym i łatwiejszym do nauki. Największa różnica tkwi w obsłudze bibliotek – dla wersji 2.x utworzono mnóstwo bibliotek, które są sukcesywnie przekładane dla wersji 3.x, jednak przełożenie wszystkich może się nie udać lub zająć lata.

xie, wystarczy przekopiować kod do wybranego komputera, a zadziała bez problemu (pod warunkiem że nie korzysta z bibliotek systemowych).
Python umożliwia też tworzenie graficznego interfejsu dla pisanych programów, programów dostępu do bazy danych, a nawet służących do łączenia się z siecią.

■ **Możliwość korzystania z bibliotek** – na kolejnych stronach dowiemy się, jak krok po kroku zainstalować środowisko programistyczne Python, a razem z nim – bardzo obszerny zbiór wbudowanych i przenośnych opcji i skryptów nazywany biblioteką standardową. Ta biblioteka pozwala na obsługę ogromnej ilości zadań programistycznych na poziomie aplikacji, jak również na dopasowywanie wzorca czy też skryptów sieciowych. Oprócz biblioteki standardowej jest bardzo dużo dodatkowych bibliotek stworzonych przez innych programistów, większość można łatwo zainstalować i wykorzystać na potrzeby swojego kodu.

■ **Integracja z innymi językami i komponentami** – warto również wiedzieć, że skrypty Pythona mogą komunikować się z innymi częściami aplikacji, w którą są wbudowane, także napisanymi w innych językach, dzięki wbudowanym mechanizmom integracji. Integracja pozwala na wykorzystanie Pythona jako narzędzia do rozbudowy aplikacji i dostosowywania ich do naszych potrzeb.
Python umożliwia wywoływanie bibliotek tworzonych dla języków C oraz C++ oraz integrację z poszczególnymi komponentami języków Java i .NET.

**Wady języka Python:**

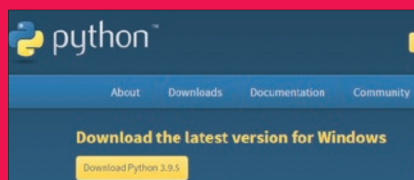
■ **Wydajność** – coś co z jednej strony jest zaletą, z innej perspektywy może być wadą. W przypadku Pythona brak konieczności kompilacji programu wynika z tego, że przy jego uruchomieniu kod źródłowy jest przekładany na format pośredni zwany kodem bajtowym, a następnie ten kod jest interpretowany. Kod bajtowy zapewnia mobilność aplikacji i nie uzależnia jej od platformy, jednak przez to, że cały kod nie jest kompilowany do tak zwanego kodu maszynowego, może się zdarzyć, że aplikacje tworzone w Pythonie będą działały znacznie mniej wydajnie niż te napisane na przykład w języku C.

■ **Dynamiczne typowanie** – ogromne ułatwienie w Pythonie, które polega na tym, że nie trzeba przypisywać typu do utworzonej zmiennej, jest również wadą, zwłaszcza gdy kod tworzy niedoświadczony programista. W wyniku szeregu operacji, które będą przetwarzać różne zmienne o początkowo niezdefiniowanych typach, może dojść do nieoczekiwanego działania kodu, a znalezienie błędu często wymaga poświęcenia wielu godzin na analizę całości.
W najnowszych wersjach Pythona można już wprowadzać tak zwane adnotacje typów, co częściowo zmniejsza tę wadę.

■ **Ułatwienia dla programisty** – Python domyślnie ukrywa pewne informacje przed programistą, aby umożliwić prosty odczyt kodu. W przypadku doświadczonych programistów ma to duży sens. Jednak w przypadku początkujących jest to często kłopotliwe, gdyż deklarując listę czy tablicę, nie zawsze można łatwo zrozumieć, dlaczego indeksowanie rozpoczyna się od 0.
Jest to jednak dość niska cena za dużą wygodę.

## PYTHON W SYSTEMACH LINUX ORAZ MAC

**Środowisko Python można również bez problemów zainstalować w systemach z rodzin Linux oraz Mac. Wystarczy skorzystać ze strony o adresie: python.org/downloads**



■ **GIL** – w Pythonie występuje GIL (Global Interpreter Lock). Jest to mechanizm, do którego może mieć dostęp jedynie jeden wątek jednocześnie, a pozostałe są blokowane. Sprawia to kłopot przy tworzeniu aplikacji wielowątkowych. Oczywiście można obejść ten problem, jednak nie jest to najlepiej rozwiązany mechanizm w tym języku.

# Przygotowanie do korzystania z Pythona w Windows

W chwili pisania tej książki najbardziej aktualną wersją Pythona jest 3.9.5 i właśnie na niej będziemy bazować we wskazówkach i bardziej zaawansowanych projektach.

## Instalujemy Pythona

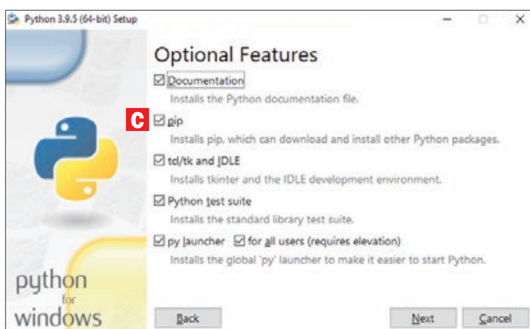
**1** Po uruchomieniu instalacji **Pythona** (**DVD-KOD: 017**) z płyty dołączonej do książki lub po pobraniu instalatora ze strony **python.org/downloads** w oknie kreatora instalacji u dołu okna zaznaczamy opcję **Add Python 3.9 to PATH** **A**. Ta opcja umożliwia szybką integrację środowiska Python z naszym systemem i korzystanie z dodatkowych modułów bez konieczności wskazywania za każdym razem dokładnej ścieżki do Pythona. Następnie klikamy na **Customize installation** **B** (jeśli mamy już zainstalowaną starszą wersję Pythona, klikamy na **Upgrade Now**).

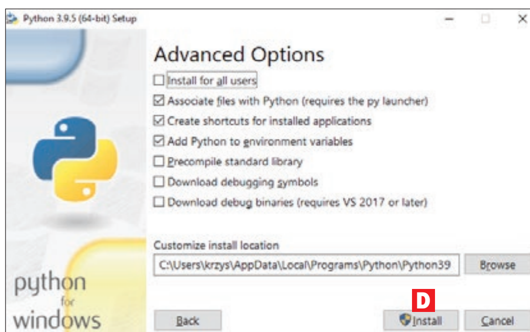
**2** W następnym kroku koniecznie zaznaczamy instalację modułu **pip** **C**, który umożliwia wygodną i szybką instalację dodatkowych bibliotek i modułów. Zaleca się zaznaczenie wszystkich opcji. Klikamy na **Next**.

**3** W kolejnym oknie upewniamy się, że jest zaznaczona opcja **Add Python to environment variables**, i klikamy na **Install** **D**.

**4** Po instalacji będziemy mogli rozpocząć korzystanie z Pythona.



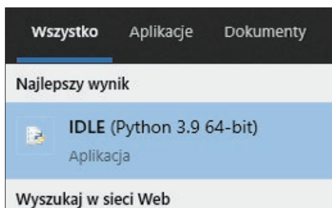
## INFORMACJE O BŁĘDACH

Korzystanie z IDLE świetnie sprawdza się przy eksperymentowaniu. Wystarczy wpisać kawałek kodu i go wykonać, a gdy pojawi się błąd, analiza jest dość prosta, gdyż nie był uruchamiany cały program, a jedynie jedna komenda.
Tutaj po wpisaniu **X** przy próbie wykonania polecenia otrzymaliśmy błąd z informacją, że nazwa „**X**" nie jest zdefiniowana. Oznacza to, że chcieliśmy wywołać zmienną, której wcześniej nie zadeklarowaliśmy.

```
>>> X
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    X
NameError: name 'X' is not defined
>>> X = 1
>>> X
1
>>> 
```

## IDLE

Najprostszym sposobem na sprawdzenie poprawności działania środowiska Pythona i samego języka jest skorzystanie z IDE (zintegrowane środowisko programistyczne) o nazwie IDLE, które jest instalowane razem z Pythonem. IDLE pozwala na interaktywne wykonywanie poleceń Pythona. W trakcie pracy w tym środowisku wykonywany jest kod i zwracane są wyniki. Jednak sam kod nie jest zapisywany w pliku. Praca w interaktywnej sesji dobrze sprawdza się przy eksperymentowaniu czy też testowaniu.
W dalszych rozdziałach będziemy pracować w bardziej rozbudowanym IDE, które umożliwia wygodne zapisywanie plików.
Jeśli jednak będziemy chcieli sprawdzić pojedyncze komendy lub zachowanie języka, najwygodniej jest to zrobić w sesji interaktywnej. Oto jak to zrobić w kilku krokach.

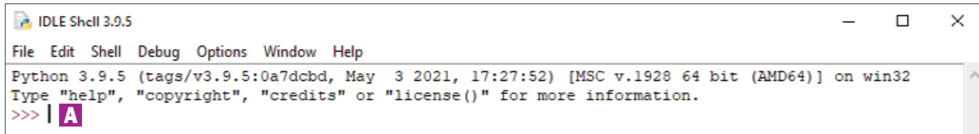



**1** W wyszukiwarkę Windows wpisujemy **IDLE** i klikamy na znalezioną pozycję.

**2** Od razu po uruchomieniu programu mamy dostęp do interaktywnej sesji dla Pythona. Zawsze wyświetlane są dwa wiersze informacyjne, a po nich w wierszu trzecim **A** możemy wpisywać nasze polecenia i je wykonywać.

**3** Polecenia wpisujemy, korzystając z klawiatury, a zatwierdzamy je, wciskając klawisz [enter]. Polecenie **print** służy do wyświetlania na ekranie informacji zawartych w nawiasie. W pierwszym przykładzie mamy do czynienia z tekstem, cały tekst musi znaleźć się pomiędzy apostrofami, wtedy zostanie poprawnie odczytany. Wewnątrz polecenia **print** możemy również wykonywać kolejne polecenia, w tym na przykład obliczenia matematyczne.

```
IDLE Shell 3.9.5
File  Edit  Shell  Debug  Options  Window  H
Python 3.9.5 (tags/v3.9.5:0a7dcbd,
Type "help", "copyright", "credits"
>>> print('Pierwsza komenda!')
Pierwsza komenda!
>>> print (2 * 8)
16
>>> print (2 ** 8)
256
>>> 
```

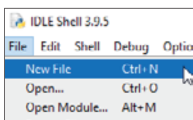


```
=============================
>>> tekst = 'Witaj świecie'
>>> tekst
'Witaj świecie'
>>> a = 8 * 8
>>> a
64
>>>
```

**4** W interaktywnej sesji możemy też deklarować zmienne i przypisywać im wartości (operator przypisania to „=”). W celu wyświetlenia wartości zmiennej należy ją wywołać, na przykład podając jej nazwę.

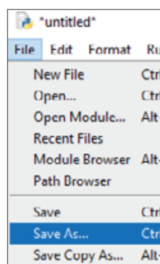
## Pierwszy skrypt w Pythonie

W celu utworzenia skryptu musimy utworzyć plik, w którym wpisany przez nas kod będzie zapisany i możliwy do uruchomienia w dowolnej chwili. W odróżnieniu od sesji interaktywnej raz utworzony skrypt możemy w każdej chwili uruchomić, a nawet rozbudować.




**1** Uruchamiamy IDLE, klikamy na górnym pasku na **File** i **New File**.

**2** Następnie wpisujemy treść skryptu **A**. Znak # umożliwia wprowadzanie komentarzy – nie są one wykonywane w trakcie działania skryptu i pełnią rolę notatek dla programisty.



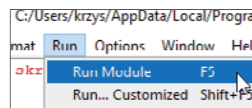

**3** Po wprowadzeniu treści skryptu klikamy na górnym pasku okna na **File**, **Save As**, a następnie nadajemy nazwę, na przykład **skrypt1.py**. Domyślnie plik zostanie zapisany w lokalizacji instalacji środowiska Python, jest to domyślna lokalizacja, w której uruchamiany jest wiersz polecenia programu IDLE w sesji interaktywnej.

**4** Po zapisaniu pliku jego lokalizacja będzie wyświetlana w tytule okna **B**.

```
*untitled*
File  Edit  Format  Run  Options  Window  Help
# Pierwszy skrypt
import sys          # Załadowanie modułu biblioteki standardowej

print(sys.version)  A
print(2 * 10)       # Wykonanie mnożenia
x = 'Test! '
print(x *5)         # Powtórzenie łańcucha znaków
```

**5** Teraz wystarczy na górnym pasku kliknąć na **Run**, **Run Module**, aby uruchomić nasz skrypt.




```
skrypt1.py  C:/Users/krzys/AppData/Local/Programs/Python/Python39/skrypt1.py (3.9.  B
File  Edit  Format  Run  Options  Window  Help
# Pierwszy skrypt
```

**6** W głównym oknie IDLE pojawi się wynik jego działania **C**.

```
============ RESTART: C:/Users/krzys/AppData/Local/Programs/Python/Python39/skrypt1.py ===========
3.9.5 (tags/v3.9.5:0a7dcbd, May  3 2021, 17:27:52) [MSC v.1928 64 bit (AMD64)]
20
Test! Test! Test! Test! Test!  C
>>>
                                                                        Ln: 48  Col: 21
```

## URUCHAMIAMY SKRYPTY W WIERSZU POLECENIA WINDOWS

Jeśli piszemy skrypty w języku Python, nie musimy korzystać z IDE do ich uruchamiania – wystarczy Wiersz polecenia w Windows. W Wierszu polecenia należy przejść do lokalizacji ze skryptem, który chcemy uruchomić, i wpisać polecenie **python [nazwa_skryptu]**, gdzie w naszym przykładzie ta nazwa to **skrypt1.py**.

```
C:\Python>python skrypt1.py
3.9.5 (tags/v3.9.5:0a7dcbd, May  3 2021, 17:27:52) [MSC v.1928 64 bit (AMD64)]
20
Test! Test! Test! Test! Test!

C:\Python>
```

## BŁĘDY PRZY BRAKU IMPORTOWANIA MODUŁU

Jeśli zamierzamy wykorzystać metodę zawartą w konkretnym module biblioteki, musimy zaimportować bibliotekę do naszego kodu, zanim z niej skorzystamy. W innym wypadku pojawi się błąd. Po zaimportowaniu możemy już korzystać z danego modułu i jego funkcji, pamiętając o bardzo prostej zasadzie, że podajemy najpierw nazwę modułu, a następnie po kropce konkretną metodę.

```
========================================== RE
>>> print(math.sqrt(4))
Traceback (most recent call last):
  File "<pyshell#22>", line 1, in <module>
    print(math.sqrt(4))
NameError: name 'math' is not defined
>>> import math A
>>> print(math.sqrt(4))
2.0
>>> |
```

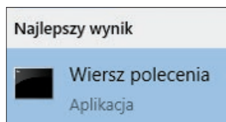
Python daje dostęp do bardzo obszernej dokumentacji, gdzie znajdziemy dokładnie opisane wraz z przykładami wszystkie funkcje dostępne w modułach, na przykład dla modułu **math** **A**: **docs.python.org/3/library/math.html**

**7** W przedstawionym skrypcie wykonaliśmy import modułu **sys**, który wykorzystaliśmy do uzyskania informacji o tym, z jakiej wersji Pythona korzystamy. Na kolejnych stronach dowiemy się, jak instalować dodatkowe biblioteki i ich moduły.

### Instalujemy dodatkowe moduły

Standardowa biblioteka, mimo że jest bardzo obszerna, nie ma w sobie wielu ciekawych i bardziej specjalistycznych modułów, które zawierają różnego rodzaju funkcje. Dzięki instalacji dodatkowych modułów możemy bardzo szybko stworzyć skomplikowane aplikacje, których napisanie całkowicie od podstaw zajęłoby wiele godzin. W tym celu skorzystamy z narzędzia **pip**, które zainstalowaliśmy razem ze środowiskiem Pythona.

**1** Uruchamiamy Wiersz polecenia Windows.




**2** Następnie wpisujemy polecenie **pip help**. Jeśli wyświetlone zostaną informacje na temat tego narzędzia, oznacza to, że instalacja była poprawna i możemy instalować dodatkowe moduły.

```
Wiersz polecenia
Microsoft Windows [Version 10.0.19041.1052]
(c) Microsoft Corporation. Wszelkie prawa zastrzeżone.

C:\Users\krzys>pip help

Usage:
  pip <command> [options]

Commands:
  install                     Install packages.
  download                    Download packages.
  uninstall                   Uninstall packages.
```

**3** W celu instalacji należy wpisać polecenie **pip install [nazwa_modułu]**, w tym przykładzie będzie to **pip install numpy** **D**.

**4** Po chwili moduł zostanie zainstalowany i będziemy mogli z niego korzystać.

```
C:\Users\krzys>pip install numpy D
Collecting numpy
  Downloading numpy-1.20.3-cp39-cp39-win_amd64.whl (13.7 MB)
     |████████████████████████████████| 13.7 MB 3.3 MB/s
Installing collected packages: numpy
Successfully installed numpy-1.20.3
WARNING: You are using pip version 21.1.1; however, version 21.1
You should consider upgrading via the 'c:\users\krzys\appdata\lo
-upgrade pip' command.

C:\Users\krzys>_
```

## SPRAWDZANIE ZAINSTALOWANYCH MODUŁÓW

W celu sprawdzenia, jakie moduły mamy zainstalowane, możemy wpisać polecenie **pip show [nazwa_modułu]**, na przykład: **pip show numpy** A. Zostanie wyświetlona informacja z wersją modułu oraz krótkim opisem.

```
C:\Users\krzys>pip list
Package    Version
---------- -------
numpy      1.20.3
pip        21.1.1
setuptools 56.0.0
```

W celu sprawdzenia wszystkich zainstalowanych modułów wpisujemy polecenie **pip list**.

```
C:\Users\krzys>pip show numpy A
Name: numpy
Version: 1.20.3
Summary: NumPy is the fundamental package for array computing with Python.
Home-page: https://www.numpy.org
Author: Travis E. Oliphant et al.
Author-email: None
License: BSD
```

W kolejnych rozdziałach zawsze, gdy będzie trzeba skorzystać z dodatkowego modułu, będzie też informacja, aby go zainstalować – każdy moduł można zainstalować przez **pip**, chyba że zostanie wskazana inna metoda.

### Korzystamy z rozbudowanego IDE – PyCharm Community

Rozbudowane środowiska programistyczne znacznie ułatwiają naukę programowania i zwiększają szybkość tworzenia kodu. W sieci można znaleźć dużo tego typu narzędzi dla Pythona. W naszych wskazówkach skorzystamy z darmowej wersji **PyCharm Community** (DVD-KOD: 013), która wystarczy nam do nauki programowania i tworzenia pierwszych skryptów, jak również rozbudowanych aplikacji. Największe zalety tego IDE to możliwość dzielenia okien, narzędzia wyszukiwania, integracja z Git i rozbudowany debugger.

**1** Uruchamiamy instalację z płyty lub pobieramy instalator ze strony **jetbrains.com/pycharm**, klikając na **Download** A, a potem znowu klikamy na **Download** B przy wersji **Community**.



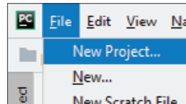

Download PyCharm

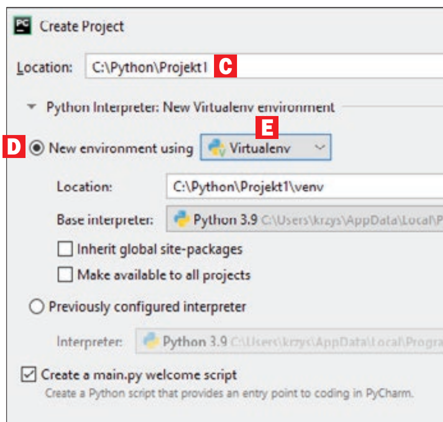
Windows | macOS | Linux

**Professional**
For both Scientific and Web Python development. With HTML, JS, and SQL support.
Download
Free trial

**Community**
For pure Python development
B
Download
Free, open-source

**2** Uruchamiamy instalator i instalujemy PyCharm, następnie uruchamiamy aplikację i klikamy na górnym pasku na **File**, **New Project**.





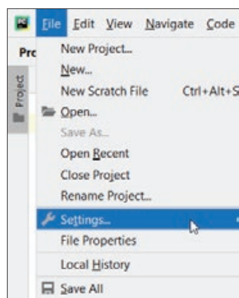

**3** Wskazujemy lokalizację **C** dla naszego nowego projektu, w którym będziemy przechowywać wszystkie skrypty tworzone w ramach nauki.

**4** Następnie zaznaczamy opcję **New environment using** **D** i z listy wybieramy **Virtualenv** **E**, na koniec klikamy na **Create** w dolnym prawym rogu.

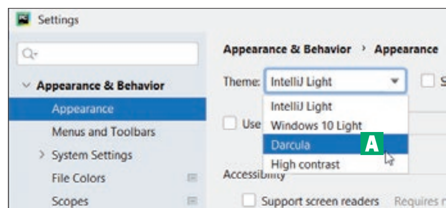
## Zmieniamy wygląd interfejsu

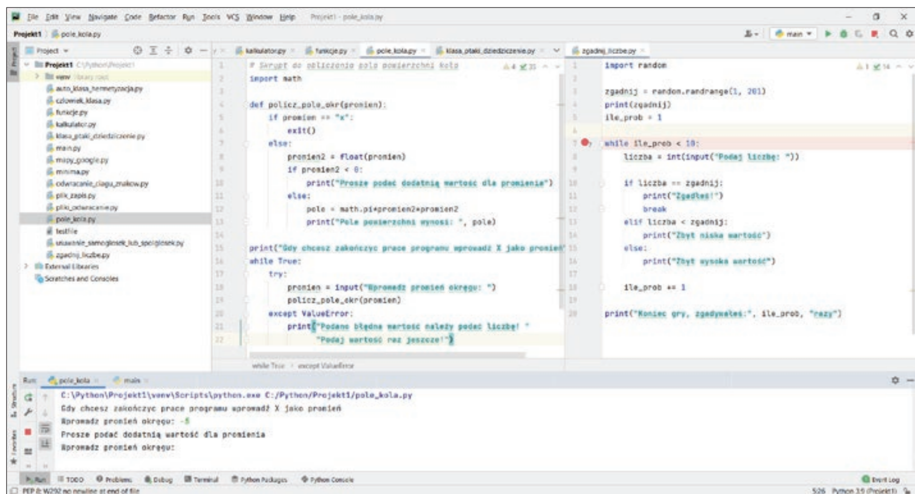
Domyślny interfejs w PyCharm ma białe tło, jeśli często pracujemy w nocy lub mamy problemy ze wzrokiem, możemy skorzystać z innych wbudowanych motywów.

**1** W celu zmiany motywu klikamy w lewym górnym rogu na **File**, **Settings**.

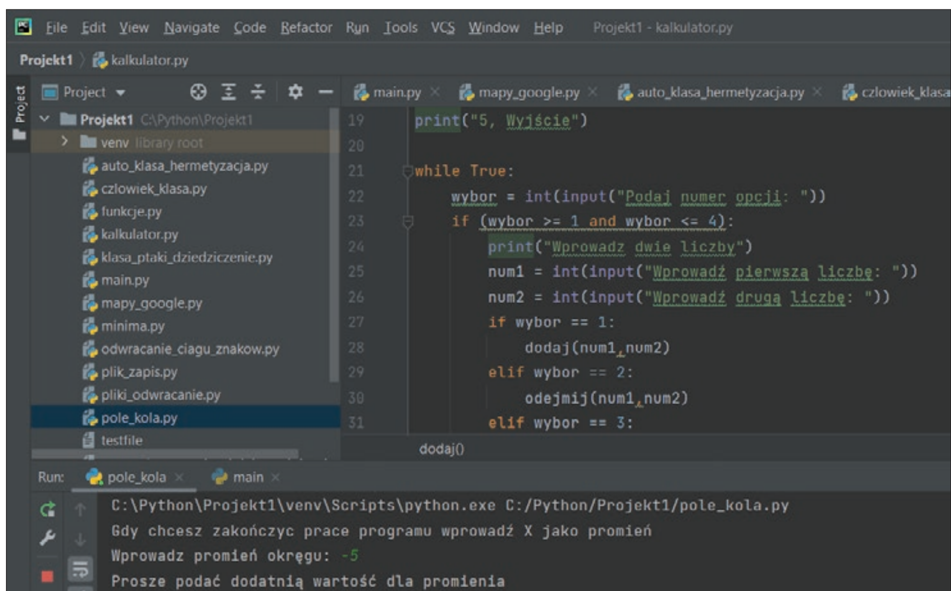



**2** Następnie klikamy po lewej stronie na liście **Appearance & Behaviour** i poniżej na



Jest to główny widok IDE PyCharm Community. Domyślnie ekran podzielony jest na trzy panele: główny z kodem naszego skryptu, dolny z terminalem lub w przypadku Windows Wierszem polecenia, gdzie możemy szybko uruchomić nasz skrypt, oraz okno po lewej stronie z wszystkimi plikami dotyczącymi naszego projektu. W dalszej części książki szczegółowo poznamy obsługę tego programu

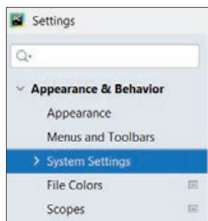



**Appearance**. Teraz po prawej stronie w polu **Theme** wybieramy jeden z motywów. Jeśli lubimy pracę w nocy i ciemne motywy – najlepiej wybrać **Darcula** **A**.

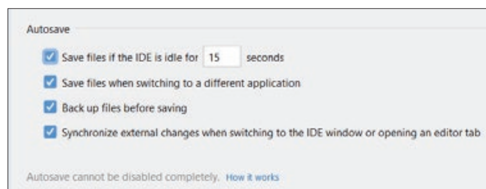
**3** Zmiany w wyglądzie interfejsu są natychmiastowe, jeśli wybrany motyw nam nie odpowiada możemy w każdej chwili zdecydować się na inny.

## Aktywujemy automatyczne zapisywanie plików

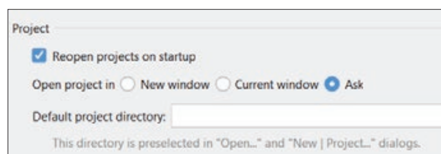
Bardzo często zdarza się, że pracujemy nad jakimś skryptem godzinami i dopiero, gdy dojdzie do awarii zasilania, zdajemy sobie sprawę, że nie zapisaliśmy swojej pracy i wszystko straciliśmy. Dlatego też warto od razu po zainstalowaniu PyCharm aktywować funkcję automatycznego zapisu danych.




**1** Ponownie wchodzimy do ustawień programu, klikając na **File**, **Settings**. Następnie na liście **Appearance & Behaviour** klikamy na **System Settings**.




**2** Teraz po prawej stronie w polu **Autosave** zaznaczamy wszystkie dostępne opcje. Dzięki temu pliki będą zapisywane, jeśli przez 15 sekund nic nie edytujemy oraz gdy aktywujemy okno innej aplikacji.
Dodatkowo w tym samym oknie możemy włączyć opcję automatycznego odtwarzania zatrzymanej pracy w projekcie. Wystarczy w polu **Project** zaznaczyć opcję **Reopen projects on startup**.

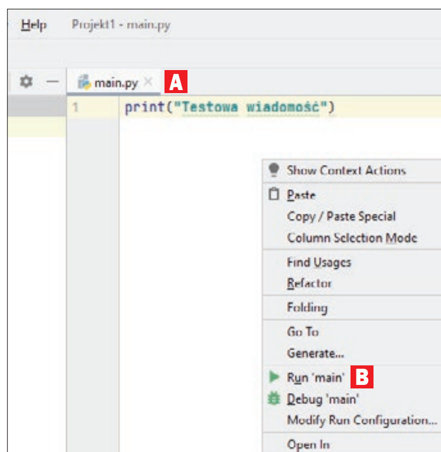


# 2 Podstawy Pythona

**W tym rozdziale poznamy podstawy Pythona. Dowiemy się między innymi, jak obsługiwać dane wejściowe i wyjściowe, jakie są typy zmiennych oraz jak je deklarować, jakie są operatory i jak korzystać z instrukcji warunkowych**

## Pierwsze kroki w PyCharm

Po utworzeniu nowego projektu możemy rozpocząć tworzenie pojedynczych skryptów, a nawet całej złożonej aplikacji. Warto zapoznać się z podstawową obsługą programu PyCharm – kilka wskazówek sprawi, że będziemy pracowali znacznie efektywniej.



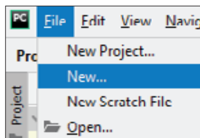

### Tworzymy skrypt i go uruchamiamy

**1** Domyślnie przy tworzeniu projektu tworzony jest też plik o nazwie **main.py** A. Możemy wyczyścić jego zawartość, zaznaczając cały tekst i go usuwając. Następnie wystarczy wprowadzić nasz kod dla danego skryptu, a później kliknąć prawym przyciskiem myszy na dowolne miejsce w głównym oknie i wybrać z menu dialogowego opcję **Run 'main'** B, gdzie **main** to nazwa skryptu.

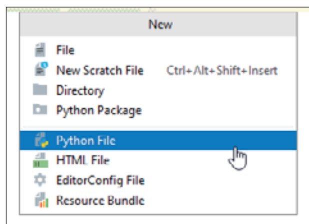



**2** W dolnej części ekranu zobaczymy, że zostaliśmy przeniesieni do zakładki **Run** C, gdzie zostanie wywołany nasz program. Po lewej stronie okna, klikając na zielony znacznik D, możemy uruchomić program ponownie. Jest to znacznie szybsze niż praca w IDLE czy Wierszu polecenia.

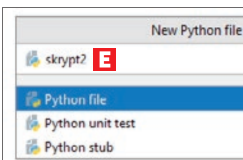
**3** Jeśli chcemy dodać kolejny skrypt do naszego projektu, klikamy na górnym pasku na **File**, **New**.

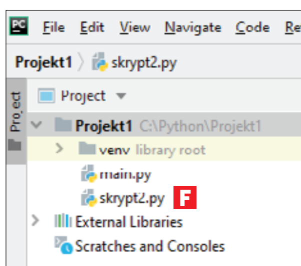



**4** Następnie klikamy na **Python File**.




**5** Podajemy nazwę dla pliku **E** i wciskamy klawisz enter. Plik zostanie domyślnie dodany do głównego katalogu naszego projektu **F** i będziemy mogli z niego korzystać.




Te podstawowe informacje pozwolą nam realizować kolejne opisywane w tej książce wskazówki. Na poszczególnych etapach będziemy poznawać obsługę kolejnych funkcji programu, które pomogą w bardziej zaawansowanych zadaniach.

## Obsługa strumieni wejścia i wyjścia

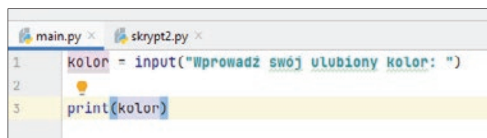
Pod tą dość skomplikowaną nazwą kryje się podstawowa funkcjonalność – odczytywanie danych od użytkownika oraz ich wyświetlanie. Do tej pory w żadnym z przykładowych skryptów nie przyjmowaliśmy danych od użytkownika. Wyświetlanie danych jest znacznie prostsze i już było prezentowane.

## Wyświetlanie danych

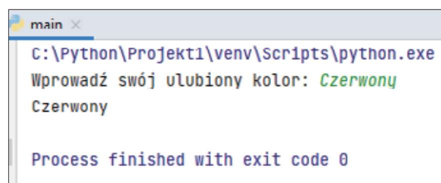
W tym celu korzystamy z polecenia **print**. Służy ono do wyświetlania w konsoli wszystkiego, co znajdzie się wewnątrz nawiasów, w przypadku zwykłego tekstu niezbędne są cudzysłowy lub apostrofy, czyli na przykład: **print("Witaj świecie")**. Wewnątrz tego polecenia możemy również wykonywać inne polecenia i wykonywać operacje matematyczne.

## Przyjmowanie danych

Tutaj proces wygląda nieco inaczej. Przyjmując dane od użytkownika, musimy móc je gdzieś przechować. W tym celu deklarujemy zmienną. Ponieważ w Pythonie typy domyślnie są dynamiczne, nie musimy wskazywać jej typu i martwić się, jakie dane wprowadzi użytkownik (co może później okazać się problematyczne). Poprawny zapis przyjęcia danych od użytkownika to na przykład **kolor = input("Wprowadź swój ulubiony kolor: ")**.



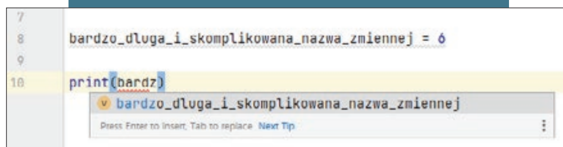

**kolor** to nazwa naszej zmiennej, **=** to operator przypisania, **input** to polecenie pozwalające na pobranie danych wprowadzanych z klawiatury przez użytkownika, a wewnątrz możemy podać treść, jaka ma być wyświetlana użytkownikowi przed wprowadzeniem danych.




Działanie skryptu jest widoczne na obrazku powyżej – widać tu, jak deklaruje się zmienne, przyjmuje i wyświetla dane. Kolorem

## WARTO WIEDZIEĆ

PyCharm domyślnie po zaznaczeniu przez nas jakiejś zmiennej podświetla wszystkie jej wykorzystania w całym kodzie.
Pracując w PyCharm, nie musimy pamiętać dokładnie wszystkich naszych zmiennych i wpisywać ich za każdym razem do skryptu. Możemy skorzystać z narzędzia podpowiedzi. Wystarczy rozpocząć wpisywanie zmiennej, a po chwili można wybrać ją z menu dialogowego narzędzia podpowiedzi.




zielonym zaznaczone są dane wprowadzane przez użytkownika. Jak widać, po wprowadzeniu tekstu został on wyświetlony w następnym wierszu, ponieważ korzystaliśmy z polecenia **print** i jako argument podaliśmy naszą zmienną **kolor**.

## Typy zmiennych i zasady ich nazywania

W powyższym przykładzie nie zadeklarowaliśmy konkretnego typu zmiennej i skorzystaliśmy z dynamicznego typowania, co oznacza, że zmienna przyjmuje typ wartości, którą przechowuje.
Czasem jednak zachodzi potrzeba dokładnego wskazania takiego typu, aby program mógł zadziałać prawidłowo.
Najważniejsze dla nas jako programisty jest to, że tworząc jakiś obiekt, na przykład **5**, przypisujemy do niego od razu zbiór operacji, jaki może być wykonywany dla danego typu. Podobnie jest w wypadku obiektu **'Test'**, który jest traktowany jako łańcuch znaków i można na nim wykonywać tylko operacje związane z łańcuchami.
Wszyscy, którzy kiedykolwiek programowali lub mieli kontakt z kodem w mniejszym lub większym stopniu, rozpoznają przedstawione w tabeli poniżej typy zmiennych. To, że Python może stosować je dynamicznie, wcale nie oznacza, że nie musimy ich znać, a wręcz przeciwnie – dobra znajomość typów oraz operacji, jakie można wykonywać na konkretnych typach, nieraz może znacznie ułatwić nam pracę.

## PODSTAWOWE TYPY ZMIENNYCH W PYTHONIE*

| TYP OBIEKTU | PRZYKŁAD |
|---|---|
| Liczby | 3.23, 101, 1e4 |
| Łańcuchy znaków | 'test', 'Adam', 'raz dwa trzy' |
| Listy | [1, [2, 'trzy'], 4] |
| Słowniki | {'jedzenie': 'arbuz', 'smak' : 'słodki'} |
| Krotki** | (1, 'arbuz', 4, 'A') |
| Pliki | myfile = open('tekst', 'dodatkowy') |
| Zbiory | set('abc'), {'a', 'b', 'c'} |
| Wartości logiczne | Wartości Boolean |
| Typy jednostek programu | Funkcje, moduły, klasy |
| Typy powiązane z implementacją | Kod skomplikowany, ślady stosu |

*Tabela Podstawowe typy zmiennych w Pythonie nie jest kompletnym zbiorem wszystkich typów, gdyż w Pythonie tak naprawdę wszystko, co przetwarzamy, jest jakimś rodzajem obiektu.
**O krotkach – patrz więcej na stronach 25..

## Liczby

Podstawowe obiekty Pythona obejmują typowe rodzaje liczb: całkowite, zmiennoprzecinkowe, a także zespolone, wymierne i inne. Liczby w Pythonie obsługują normalne działania matematyczne; dokładną listę operatorów znajdziemy w ramce na stronie obok. Przedstawione w tej ramce operatory arytmetyczne łatwiej można zrozumieć w działaniu na liczbach w przykładowym skrypcie, dlatego warto z nimi ekspe-

## OPERATORY ARYTMETYCZNE

| TYP OBIEKTU | PRZYKŁAD |
|---|---|
| + | Dodawanie |
| - | Odejmowanie |
| * | Mnożenie |
| ** | Podnoszenie do potęgi |
| / | Dzielenie |
| % | Reszta z dzielenia (modulo) |
| // | Dzielenie całkowite |

rymentować – im więcej kodu przepiszemy i uruchomimy sami w IDE na naszym komputerze, tym szybciej zapamiętamy zasady jego działania i programowania.

Jak widać na dwóch kolejnych ilustracjach, bez żadnych dodatkowych deklaracji konkretnego typu wszystkie liczby zostały odpowiednio przetworzone.

Próba wykonania tych działań w języku na przykład C++ zmusiłaby nas od razu do przemyślenia, jakie konkretnie działania mają być wykonywane, jakich wyników się spodziewamy i dopiero na tej podstawie moglibyśmy określić typy, na przykład zmiennoprzecinkowe, całkowite itp. Python jest znacznie mniej wymagający pod tym względem.

main.py × skrypt2.py ×

```
a = 21
b = 5
c = 0

c = a + b #Dodawanie
print(c)
c = a - b #Odejmowanie
print(c)
c = a * b #Mnożenie
print(c)
c = a ** b #Podnoszenie do potęgi
print(c)
c = a / b #Dzielenie
print(c)
c = a % b #Modulo
print(c)
c = a // b #Dzielenie całkowite
print(c)
```

main ×

```
C:\Python\Projekt1\venv\Scripts\pyt
26
16
105
4084101
4.2
1
4

Process finished with exit code 0
```

### Łańcuchy znaków

Najprościej możemy określić łańcuchy znaków jako słowa lub kawałki tekstu. W języku Python łańcuchy są traktowane jako sekwencje, możemy korzystać z operacji, które zakładają uporządkowane pozycjonowanie

## WARTO WIEDZIEĆ

Jeśli chcemy uzyskać informacje na temat liczby cyfr składających się na daną liczbę, możemy zastosować prosty trik:

```
>>> print(len(str(2 ** 1000)))
302
>>>
```

Skorzystaliśmy tutaj z polecenia **len**, które służy do wskazywania długości łańcucha znaków, gdyż zadeklarowaliśmy wynik podnoszenia do potęgi 1000 liczby 2 jako ciąg znaków, a nie liczbę. Wynikiem takiej operacji jest liczba znaków, czyli cyfr, które składają się na wynik.

Korzystając z narzędzia debugowania w PyCharm, możemy prześledzić, jak zmieniają się wartości poszczególnych zmiennych. Więcej o tym narzędziu dowiemy się w kolejnych, bardziej zaawansowanych rozdziałach.

main.py × skrypt2.py ×

```
a = 2 ** 1000   a: 10715086071862673209484
b = len(str(a))   b: 302
print(b)
```

```
>>> S = "Tekst"
>>> len(S)
5
>>> S[0]
'T'
>>> S[4]
't'
>>> S[5]
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    S[5]
IndexError: string index out of range
>>>
```

elementów. Jeśli mamy łańcuch **"Tekst"** (ilustracja powyżej), możemy zweryfikować jego długość i odnosić się do każdego znaku osobno, korzystając z indeksowania.
Warto zwrócić uwagę, że indeksowanie zawsze zaczyna się od **0**, a nie od **1**. Oznacza to, że łańcuch o długości 5 jest zaindeksowany w tablicy od 0 do 4, a każde odwołanie poza obszar tej tablicy będzie skutkowało błędem, gdyż nigdy nie możemy odwoływać się do elementów nieistniejących. Możemy odwoływać się natomiast do elementów pustych.

## WARTO WIEDZIEĆ

W Pythonie można indeksować od tyłu – jest to szczególnie przydatne w zadaniach wymagających odczytywania poszczególnych znaków z długich łańcuchów. Nie musimy znać konkretnego rozmiaru łańcucha, aby odwołać się do jego ostatniego elementu. Wystarczy skorzystać z takiego zapisu **A**.
Druga forma **B** jest również poprawna, jednak znacznie dłuższa. Korzystając z ujemnego indeksowania, możemy odwoływać się do kolejnych znaków od końca.

```
>>> S[-1]          A
't'
>>> S[len(S)-1]    B
't'
>>>
```

## WARTO WIEDZIEĆ

W Pythonie wszędzie, gdzie występuje nawias, możemy użyć dowolnego wyrażenia, nie musi być to liczba. Często przydatne jest na przykład stosowanie tak zwanych wycinków.
Zapis **S[1:4]** **A** pozwala na wyświetlanie znaków pomiędzy 1 a 4 indeksem.
Łańcuchy tak samo jak pozostałe sekwencje obsługują również konkatenację, czyli łączenie dwóch łańcuchów poprzez wykorzystanie operatora **+** **B**, oraz powtórzenie **C**, gdzie możemy budować nowy łańcuch, wykorzystując stary.
**Uwaga!** Zastosowanie znaku **+** w celu dodawania dwóch liczb jest zupełnie inną operacją niż konkatenacja w przypadku łączenia łańcuchów. Stosowanie tego samego operatora do różnych zadań jest możliwe dzięki temu, że Python ma właściwość zwaną polimorfizmem. Należy również pamiętać o zasadzie niezmienności, która dotyczy łańcuchów. W dużym skrócie: mimo że łańcuchy są indeksowane, nie możemy, odwołując się do konkretnego indeksu, przypisywać im nowej wartości – nie są to tablice. Jeśli chcemy zmienić łańcuch, musimy utworzyć nowy obiekt. W teorii wydaje się to skomplikowane, jednak w rzeczywistości jest dość proste, co widać na ilustracji poniżej.

```
>>> S
'Tekst'
>>> S[1:4]    A
'eks'
>>>
```

```
>>> S
'Tekst'
>>> S + 'xyz'    B
'Tekstxyz'
>>> S * 5    C
'TekstTekstTekstTekstTekst'
>>>
```

```
'Tekst'
>>> S[0] = 'A'
Traceback (most recent call last):
  File "<pyshell#24>", line 1, in <module>
    S[0] = 'A'
TypeError: 'str' object does not support item assignment
>>> S = 'A' + S[1:]
>>> S
'Aekst'
>>>
```

## NAZWY ZMIENNYCH

Zmienne możemy nazywać praktycznie dowolnie wedle naszego uznania. Istnieje jednak grupa nazw, których nie możemy użyć, oraz kilka podstawowych zasad, których musimy przestrzegać. Przede wszystkim w nazwie zmiennej nie może być spacji. Jeśli chcemy, aby nazwa składała się z osobnych wyrazów, używajmy znaku "_" do łączenia słów. Nazwa zmiennej nie może też rozpoczynać się od cyfry ani nie może zawierać znaków specjalnych, takich jak **$**. Nie może również należeć do zbioru słów zastrzeżonych, który zawiera operatory i strukturę języka ze słowami kluczowymi, na przykład **class**, **break**, **if** itp. Takich słów jest około 30.
Jeżeli po nadaniu nazwy zmiennej otrzymujemy błąd przy uruchomieniu programu, należy sprawdzić, czy nazwa jest poprawna.

# Instrukcje warunkowe

Przedstawione na poprzednich stronach przykłady były bardzo proste i bazowały jedynie na pojedynczych linijkach kodu. Jeśli jednak chcemy stworzyć bardziej złożone skrypty, w każdym z nich na pewno znajdą się instrukcje warunkowe. Tego typu instrukcje w Pythonie rozpoczynamy od słowa kluczowego **if**.
W dużym uproszczeniu: instrukcja **if** służy do wyboru alternatywnych działań na podstawie wyniku testu logicznego. Tego typu instrukcje możemy dowolnie zagnieżdżać w sobie lub dodawać kolejne alternatywne wybory. Ogólna postać takiej instrukcji wygląda zawsze w podobny sposób jak w przykładzie **A**.
Po instrukcji **if** następuje test logiczny, może być rozbudowany i umieszczony w nawiasach, a po nim zawsze następuje znak "**:**". Następnie w kolejnym wierszu jest wcięcie i blok operacji, które zostaną wykonane, jeśli wynik logiczny testu będzie pozytywny. Po nim w naszym przykładzie jest instrukcja **elif**, jest ona opcjonalna i służy do wprowadzania kolejnych testów logicznych na tym samym poziomie. Następnie mamy instrukcję **else**, jest to również instrukcja opcjonalna, jednak występuje praktycznie zawsze, jeśli korzystamy z instrukcji **if**; kod umieszczony w jej bloku jest wykonywany zawsze wtedy, gdy wszystkie testy w ramach instrukcji **if** lub **elif** zwróciły wynik fałszywy. Należy zwrócić szczególną uwagę na znak "**:**", który kończy każdy wiersz z instrukcjami **if**, **elif**, **else**, oraz na wcięcia w kodzie.

**A**

Project: Projekt1 C:\Python\Projekt1 — venv library root, main.py, skrypt2.py, External Libraries, Scratches and Consoles

main.py | skrypt2.py

```python
a = 5
b = 4
c = 3

if a < b:
    print(a)
elif b < c:
    print(b)
else:
    print(c)
```

Run: skrypt2

```
C:\Python\Projekt1\venv\Scripts\python.exe C:/Python/Projekt1/skrypt2.py
3

Process finished with exit code 0
```

## UWAGA! WCIĘCIA!

W Pythonie niezwykle istotną rolę pełnią wcięcia w tekście – tworzonych skryptów nie formatujemy ze względów estetycznych, ale dlatego, że jest to konieczne do ich działania. Składnia Pythona wymusza stosowanie odpowiednich wcięć w celu rozróżnienia bloków kodu, które mają być wykonywane po sobie. Jeśli dodajemy kolejne poziomy instrukcji warunkowych, musimy zastosować kolejne poziomy wcięć.

```
main.py   skrypt2.py
a = 3
b = 9
c = 5

if a < b:
    a = a + 5
    if a < b:
        print(b)
    print(a)
elif b < c:
    print(b)
else:
```

```
ekt1\venv\Scripts\python.exe C:/Python/Projekt1/skrypt2.py
```

## OPERATORY PORÓWNANIA I LOGICZNE

Wcześniej poznaliśmy operatory arytmetyczne, które służą do wykonywania operacji z liczbami i zmiennymi. W Pythonie jest bardzo dużo różnego typu operatorów, drugą najważniejszą grupą są operatory porównania, które bardzo często wykorzystujemy w instrukcjach warunkowych, pętlach i innego typu sytuacjach, w których jest konieczne porównanie wartości dwóch lub większej liczby elementów.

```
main.py   skrypt2.py
a = 3
b = 7
c = 5

if a < b and c < b:
    a = a + 5
    if a > b or b > c:
        if not(b < c):
            print(c)
        print(b)
    print(a)
else:
if a < b and c < b
```

Przykład instrukcji warunkowych z wykorzystaniem różnych operatorów porównania oraz logicznych

| ZNAK | OPIS |
|---|---|
| **==** | Sprawdzenie równości (nie mylić z przypisaniem, czyli =); A == B zwróci fałsz* |
| **!=** | Nierówne; A != B zwróci prawdę |
| **<>** | Nierówne; A <> B zwróci prawdę |
| **>** | Większe; A > B zwróci fałsz |
| **<** | Mniejsze; A < B zwróci prawdę |
| **>=** | Większe lub równe; A >= B zwróci fałsz |
| **<=** | Mniejsze lub równe; A <= B zwróci prawdę |
| **and** | i – zwraca prawdę, jeśli dwa argumenty są prawdziwe |
| **or** | lub – zwraca prawdę, jeśli jeden z argumentów jest prawdziwy |
| **not** | nie – zwraca prawdę, jeśli warunek nie jest spełniony |

*Przyjmujemy, że A = 10, B = 20.

# Pętle

**W** Pythonie korzysta się głównie z dwóch instrukcji w odniesieniu do pętli – pętli **while** oraz pętli **for**. Ogólnie pętle służą do wykonywania instrukcji powtarzających jakieś działanie. Pierwsza – **while** – umożliwia zapisywanie w kodzie uniwersalnych pętli, druga – **for** – umożliwia wykonywanie bloków kodu w sekwencji dla każdego elementu.

## Pętla while

Instrukcja **while** w Pythonie to najbardziej uniwersalna konstrukcja iteracyjna dla tego języka. W dużym uproszczeniu: pozwala ona na powtarzanie wykonywania bloku kodu, dopóki test znajdujący się przy deklaracji pętli zwraca wartość logiczną będącą prawdą. Po spełnieniu warunku program jest realizowany dalej poza blokiem pętli. Jeśli od samego początku test umieszczony w nagłówku pętli będzie zwracał fałsz, pętla **while** nie zostanie wykonana ani razu.
Przyjrzyjmy się teraz kilku pętlom **while** w różnych zastosowaniach:

A

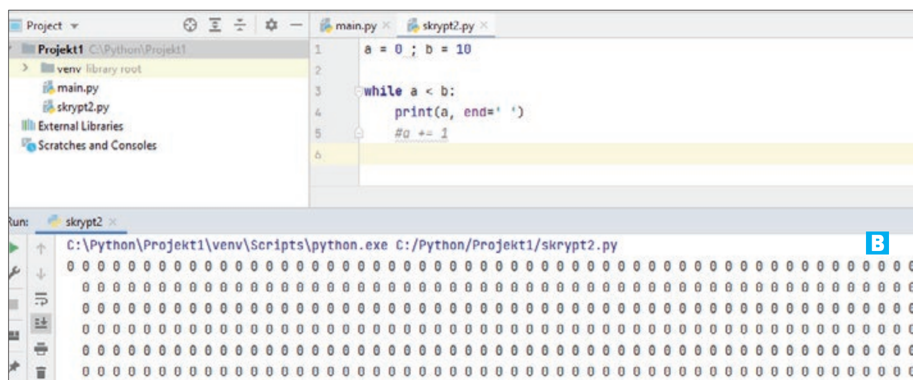
```
>>> a=0; b=10
>>> while a < b:
        print(a, end=' ')
        a += 1

0 1 2 3 4 5 6 7 8 9
>>>
```

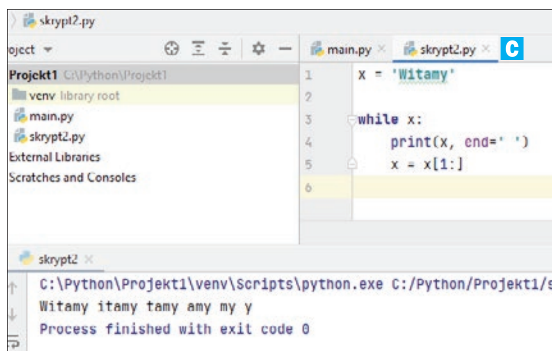
Tutaj mamy klasyczny przykład wykorzystania pętli **while** A jako pętli iteracyjnej. Przeważnie znacznie szybciej ten sam efekt można osiągnąć za pomocą pętli **for**, co pokażemy w przykładach na kolejnych stronach. Po zadeklarowaniu zmiennych i przypisaniu im wartości przechodzimy do pętli **while**, która w nagłówku ma zdefiniowany test logiczny, następnie wewnątrz pętli wykonujemy różne operacje, między innymi zwiększając wartość zmiennej **a**, tak że po pewnym czasie warunek z nagłówka zostaje spełniony, a my wychodzimy z pętli.
Jeśli warunek wyjścia z pętli nie będzie mógł być spełniony, pętla będzie działać w nieskończoność B lub do wyczerpania zasobów

### UWAGA! NIESKOŃCZONE PĘTLE!

Istnieje również możliwość utworzenia tak zwanej nieskończonej pętli, to znaczy pętli, której warunek logiczny nigdy nie będzie spełniony. Należy uważać, tworząc tego typu pętle, gdyż jeśli nie umieścimy w nich kodu pozwalającego na opuszczenie pętli, nasz skrypt będzie pracował bez przerwy, dopóki go nie wyłączymy bezpośrednio, może nawet doprowadzić do zajęcia całej pamięci operacyjnej komputera.



naszego urządzenia. Po uruchomieniu w IDLE taką pętlę można przerwać, korzystając z kombinacji Ctrl+C, a w PyCharm – korzystając z Ctrl+F2.
W tym przykładzie **C** w nagłówku został umieszczony warunek, który sprawia, że pętla będzie wykonywana, dopóki łańcuch nie będzie pusty. Wewnątrz pętli odcinamy po jednym znaku przy każdej iteracji.




## Instrukcje break, continue, pass, else w pętlach

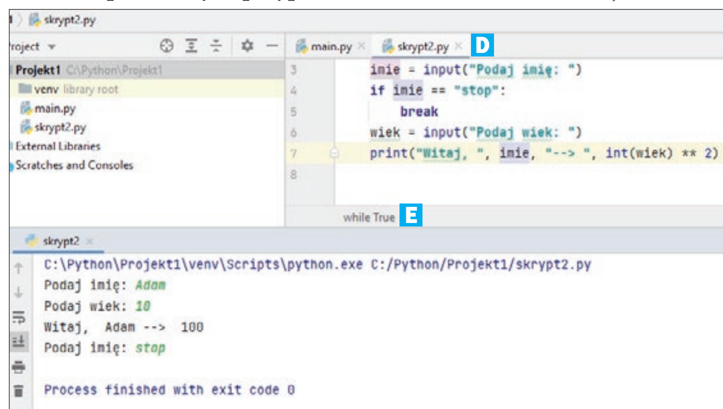
Zanim przejdziemy do pisania zaawansowanych skryptów, musimy koniecznie zapoznać się z tymi instrukcjami:

- **break** – powoduje wyjście z najbliższej obejmującej daną instrukcję pętli (omija całą instrukcję);
- **continue** – przechodzi na górę najbliższej obejmującej daną instrukcję pętli (do wiersza nagłówka);
- **pass** – nie wykonuje żadnej akcji, jest pustym pojemnikiem instrukcji;
- **else w bloku pętli** – wykonywana jest tylko i wyłącznie wtedy, gdy pętla kończy się normalnie, to znaczy bez natrafienia na instrukcję **break**.

A oto przykłady dla przedstawionych powyżej instrukcji (bez **pass** używanej w bardziej skomplikowanych przypadkach).

### Instrukcja break

Ta instrukcja powoduje natychmiastowe wyjście z pętli. Dzięki niej kod umieszczony po niej nie zostanie wykonany, pozwala to na zredukowanie stopnia zagnieżdżenia skomplikowanych przypadków. Dzięki temu możemy utworzyć nieskończone pętle interaktywne, które zostaną zakończone wtedy, gdy użytkownik o tym zdecyduje.
W tym przykładzie **D** w nagłówku pętli **while** wpisany jest warunek **True** **E**, oznacza to, że pętla będzie wykonywana bez przerwy, dopóki nie trafi na instrukcję przerywającą, w tym przypadku **break**. Pokazany w przykładzie kod działa w nieskończonej pętli i prosi użytkownika o podanie imienia oraz wieku, a następnie wyświetla te informacje. W celu zatrzymania skryptu należy podać jako imię – **stop**. Wtedy zostanie wywołana instrukcja **break**. Jest to prosty przykład, ale w rzeczywistości bardzo często stosuje się **break** właśnie w przypadku zagnieżdżonych instrukcji warunkowych.




### Instrukcja continue

Ta instrukcja powoduje natychmiastowe przejście na górę pętli. Czasami pozwala to uniknąć różnego typu zagnieżdżeń. Możemy

również wykorzystywać ten kod do pomijania pewnych iteracji, na przykład jeśli przy importowaniu danych z pliku nie chcemy importować jakiegoś wiersza, ponieważ nie spełnia on określonych warunków, możemy go łatwo pominąć dzięki instrukcji **continue**. W naszym przykładzie skorzystamy z tej instrukcji do pomijania wyświetlania liczb nieparzystych.

Skorzystanie z **continue** F wewnątrz instrukcji **if** powoduje przeskoczenie do kolejnej iteracji z pominięciem **print** na końcu kodu.

Takie wykorzystanie instrukcji **continue** jest bardzo podobne do instrukcji **goto** znanej z innych języków programowania. W Pythonie nie ma takiej instrukcji, ale wykorzystując w odpowiedni sposób **continue**, możemy uzyskać podobny efekt.

## Instrukcja else w pętli

W przypadku połączenia z częścią pętli **else** instrukcja **break** często pomaga w eliminowaniu potrzeby używania flagi (opcji) statusu wyszukiwania z innych języków. W przykładowym fragmencie kodu G skorzystamy z instrukcji **else** oraz **break** w celu ustalenia liczby pierwszej.

G

```
x = 10 ; y = 13
x = y // 2
while x > 1:
    if y % x == 0:
        print(y, 'Nie jest liczbą pierwszą, ma czynnik', x)
        break
    x -= 1
else:
    print(y, 'Jest liczbą pierwszą')
```

```
C:\Python\Projekt1\venv\Scripts\python.exe C:/Python/Projekt1/skrypt2.py
13 Jest liczbą pierwszą

Process finished with exit code 0
```

H

```
x = 10 ; y = 15
x = y // 2
while x > 1:
    if y % x == 0:
        print(y, 'Nie jest liczbą pierwszą, ma czynnik', x)
        break
    x -= 1
else:
    print(y, 'Jest liczbą pierwszą')
```

```
C:\Python\Projekt1\venv\Scripts\python.exe C:/Python/Projekt1/skrypt2.py
15 Nie jest liczbą pierwszą, ma czynnik 5

Process finished with exit code 0
```

F

```
x = 10
while x:
    x = x -1
    if x % 2 != 0:
        continue
    print(x, end=' ')
```

```
C:\Python\Projekt1\venv\Scripts\python.exe C:/
8 6 4 2 0
Process finished with exit code 0
```

Celem działania tego kodu jest sprawdzenie, czy **y** jest liczbą pierwszą. Należy zwrócić uwagę na lokalizację instrukcji **else** – jest ona umieszczona poza pętlą na tym samym poziomie wcięcia co pętla **while**. Oznacza to, że jeśli warunek wejściowy z nagłówka nie zostaje spełniony lub pętla w ramach wszystkich iteracji nie trafi na instrukcję **break**, zostanie wykonany kod zawarty w instrukcji **else**. W naszym przykładzie pozwala to na założenie, że liczba **y** jest liczbą pierwszą, ponieważ nie trafiliśmy na instrukcję **break**. Trzeba tylko pamiętać, że przykład H jest prostym rozwiązaniem, który ma za zadanie zobrazować możliwości instrukcji **else**, wskazywanie liczby pierwszej nie jest realizowane idealnie i dla każdego przypadku.

# Pętle for

W Pythonie używa się pętli **for** głównie jako uniwersalnego iteratora po sekwencjach. Dzięki niej możemy przechodzić elementy w dowolnym obiekcie, który jest uporządkowaną sekwencją (na przykład łańcuch znaków, listy, krotki itp.). Pętla **for** rozpoczyna się od wiersza nagłówka, który musi określać cel lub cele przypisania wraz z obiektem, przez który zamierzamy iterować; po nagłówku znajduje się blok, przez który będziemy przechodzić w ramach pętli. Instrukcje **else**, **break**, **continue** działają w pętli **for** w sposób analogiczny do pętli **while**.

```
A
for i in ["test","Adam","Piotr","Cześć"]:
    print(i)

C:\Python\Projekt1\venv\Scripts\python.exe C:/Python/Projekt1
test
Adam
Piotr
Cześć

Process finished with exit code 0
```

Przyjrzyjmy się, jak działa pętla **for**, na kilku prostych przykładach.

Jak widać, zmienna **i** w tym przykładzie **A** jest celem, warto zwrócić uwagę, że nie musimy jej wcześniej zadeklarować. Ta zmienna przy każdej iteracji przyjmuje kolejno wartości wskazanego obiektu – w tym wypadku jest to lista z czterema elementami. Pętla kończy działanie, gdy skończą się elementy wskazanego obiektu.

```
B
for i in range(10):
    print(i, end=' ')

C:\Python\Projekt1\venv\Scripts\python.exe C:/
0 1 2 3 4 5 6 7 8 9
Process finished with exit code 0
```

W tym przykładzie **B** skorzystaliśmy z wbudowanej funkcji **range**, która umożliwia wskazanie zakresu iteracji. Może on przyjąć aż trzy argumenty, a jeśli podamy tylko jeden, będzie to argument wskazujący rozmiar indeksowanego zakresu. Jeżeli zatem podamy **10**, nasz zakres to od 0 do 9. Domyślnie wartością początkową dla indeksowania jest 0, a krok inkrementacji wynosi 1. Możemy również sami zdefiniować początek i koniec zakresu, przekazując do funkcji **range** dwa argumenty. Pierwszy informuje o początku, a drugi o końcu zakresu.

```
for i in range(1,11):
    print(i, end=' ')

C:\Python\Projekt1\venv\Scripts\python.exe C:/
1 2 3 4 5 6 7 8 9 10
Process finished with exit code 0
```

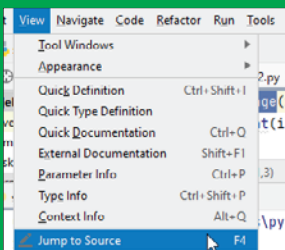
Dlaczego więc ostatnia wyświetlona liczba to 10 **C**, skoro zdefiniowaliśmy koniec zakresu jako 11? Jest to związane z zasadą działania funkcji – w skrócie dla Pythona wskazanie argumentu **stop** w tej funkcji powoduje wyświetlenie przedostatniej wartości, a gdy iterator dojdzie do wskazanej przez nas war-

```
D
for i in range(1,11,3):
    print(i, end=' ')

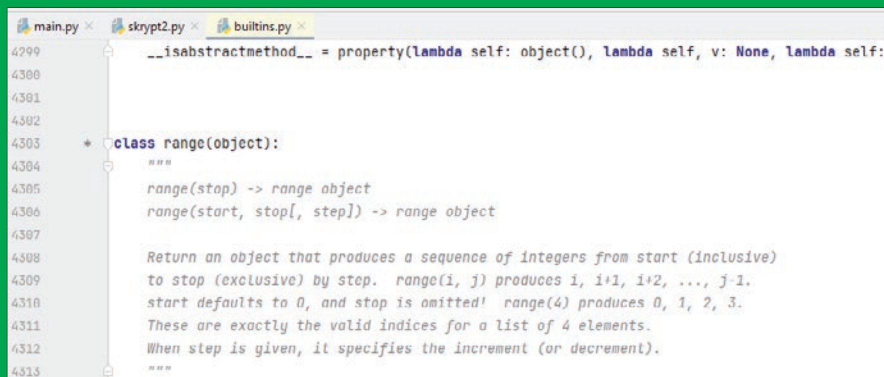
C:\Python\Projekt1\venv\Scripts\python.exe C:/P
1 4 7 10
Process finished with exit code 0
```

## SZCZEGÓŁY FUNKCJI

Możemy oczywiście szukać informacji w dokumentacji Pythona, w różnych źródłach internetowych i książkach. Jednak jeśli korzystamy z IDE PyCharm, możemy bardzo szybko samemu sprawdzić zasady działania poszczególnych funkcji i to nie tylko tych wbudowanych. Wystarczy, że po wpisaniu nazwy funkcji zaznaczymy ją, a następnie wciśniemy klawisz F4 lub klikniemy na górnym pasku na **View** i na **Jump to source**.



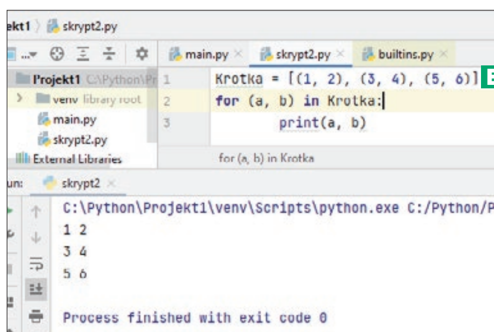

Po prawej stronie otwarty zostanie plik zawierający opis funkcji lub klasy, która nas interesuje.
W przypadku wbudowanych funkcji i klas Pythona znajdziemy przy jej deklaracji kompletny opis działania. W przypadku **range** znajduje się tu informacja o tym, że argument stop działa do przedostatniej wartości (**j-1**).




tości, zatrzyma pracę i już nic nie wyświetli. Jest to powodem częstych błędów wśród początkujących.
W przykładzie **D** podaliśmy do funkcji **range** trzy argumenty: pierwszy jest punktem startu, drugi to zatrzymanie odliczania -1, a ostatni to opcjonalny argument, który służy do wskazania stopnia inkrementacji.

### Przypisywanie krotek

Wcześniej przeczytaliśmy o tym, że istnieje wiele różnych typów w Pythonie. Jednym z tych typów są krotki. W dużym uproszczeniu krotki są w przybliżeniu listą, której nie można modyfikować. Ta cecha sprawia, że są sekwencjami niezmiennymi i czasem zachodzi potrzeba wykorzystania ich zamiast zwykłej listy. Ciekawą interakcję możemy wykonywać przy zastosowaniu krotek w pętlach **for**, gdyż możliwe jest zdefiniowanie celu jako krotki celów. Dzięki temu może-



my przechodzić przez krotkę w dowolny sposób uzależniony tylko i wyłącznie od wskazanego celu.
Po zdefiniowaniu krotki **E** (na poprzedniej stronie) i poznaniu jej struktury możemy dowolnie dostosować cel pętli **for**, aby przyjmował odpowiednią liczbę parametrów jako cel.

## Zagnieżdżone pętle for

Każda pętla for może być zagnieżdżona w kolejnej, dzięki temu możemy sortować różnego typu zakresy lub wykonywać skomplikowane operacje porównania.
Mamy dwie listy obiektów – **dane** oraz **test** **F**. Skrypt przeszukuje listę **dane**, by znaleźć obiekty z listy **test**. Zagnieżdżanie pętli **for** w kolejnej pętli **for** pozwala na przeszukiwanie z porównaniem dla wszystkich elementów wskazanych list.
Korzystając z operatora **in**, możemy nieco uprościć nasz przykładowy skrypt **G**. Korzystając z operatora **in**, mamy możliwość w sposób niejawny przeszukiwać obiekty, szukając dopasowania. Dzięki temu możemy zastąpić jedną z pętli. Takie zastosowanie operatora **in** warto zapamiętać, gdyż możemy dzięki niemu znacznie uprościć sobie kodowanie.

**F**

```
dane = ["aba", 101, (5, 6), 2.22]
test = [(5, 6), 3.22]
for klucz in test:
    for rzecz in dane:
        if rzecz == klucz:
            print(klucz, "znaleziono")
            break
    else:
        print(klucz, "nie znaleziono")
```

```
C:\Python\Projekt1\venv\Scripts\python.exe C:/Python/Projekt1/skrypt2.py
(5, 6) znaleziono
3.22 nie znaleziono

Process finished with exit code 0
```

**G**

```
dane = ["aba", 101, (5, 6), 2.22]
test = [(5, 6), 3.22]
for klucz in test:
    if klucz in dane:
        print(klucz, "znaleziono")
else:
    print(klucz, "nie znaleziono")
```

```
C:\Python\Projekt1\venv\Scripts\python.exe C:/Python/Projekt1/skrypt2.py
(5, 6) znaleziono
3.22 nie znaleziono

Process finished with exit code 0
```

# Ćwiczenia związane z pętlami

Zanim przejdziemy do programowania, warto przećwiczyć zdobyte umiejętności. Najpierw zapoznajmy się z treścią ćwiczeń i postarajmy się samemu dojść do rozwiązania, a dopiero potem sprawdźmy gotowe na stronie obok, porównując je z naszym.
**Uwaga!** Każde zadanie może być rozwiązane na kilka sposobów.

**1** Napisz skrypt, który wyświetli liczby parzyste z zakresu od 0 do 10.

**2** Napisz skrypt, który wypisze co czwartą liczbę z zakresu od 1 do 50.

**3** Napisz skrypt, który znajdzie najmniejszą wartość na liście.

**4** Napisz skrypt, który poda sumę liczby aktualnej i poprzedniej z zakresu od 1 do 10.

```
for i in range (1,11):
    if i % 2 == 0:
        print(i)
```

```
C:\Python\Projekt1\venv\Scripts\python.
2
4
6
8
10
```

1

## Rozwiązanie 1

W rozwiązaniu wykorzystaliśmy pętlę **for**. Kluczem jest wybranie odpowiedniego zakresu i przekazanie go za pomocą funkcji **range**. A do określenia, czy liczba jest parzysta, czy nie, wewnątrz pętli utworzyliśmy warunek logiczny, który sprawdza, czy reszta z dzielenia konkretnej wartości przez 2 jest równa; jeśli jest, to liczba jest parzysta i powinna zostać wyświetlona.

## Rozwiązanie 2

Kluczem do prostego rozwiązania jest pamiętanie, że funkcja **range** przyjmuje również trzeci parametr, który jest krokiem iteracji. Jeśli nie zdefiniujemy tego parametru, funkcja będzie przechodzić o wartość 1. Bez wykorzystania tej zależności rozwiązanie zadania staje się znacznie bardziej rozbudowane.

## Rozwiązanie 3

Tu mamy dwa rozwiązania – pierwsze z wykorzystaniem pętli i drugie, które wykorzystuje wbudowane w Pythona funkcje i nie ma bezpośrednio nic wspólnego z pętlami.

Pierwsze polega na zdefiniowaniu listy z liczbami i utworzeniu zmiennej pomocniczej, która służy do zapamiętywania najmniejszej wartości. Wewnątrz pętli przechodzimy przez całą listę, sprawdzając w warunku logicznym, czy dana wartość jest najmniejsza, porównując ją z kolejnymi elementami listy. Na koniec wyświetlamy najmniejszą wartość.

Drugie rozwiązanie zajmuje tylko dwa wiersze: w jednym tworzymy listę, a w drugim wyświetlamy jej najmniejszą wartość, korzystając z wbudowanej w Pythona funkcji **min** do pozyskiwania najmniejszego elementu listy.

```
for i in range (1,50,4):
    print(i)
```

```
C:\Python\Projekt1\venv\Scripts\python
1
5
9
13
17
21
25
29
33
```

2

```
lista = [1, 3, 7, 11, 2, -6, 0]

najmniejsza = lista[0]
for i in lista:
    if najmniejsza > i:
        najmniejsza = i

print("Najmniejsza liczba to:", najmniejsza)
```

```
C:\Python\Projekt1\venv\Scripts\python.exe C:/Python/Pr
najmniejsza liczba to: -6

Process finished with exit code 0
```

3A

```
lista = [1, 3, 7, 11, 2, -6, 0]

print("Najmniejsza liczba to:", min(lista))
```

```
C:\Python\Projekt1\venv\Scripts\python.exe C:/Python/Pr
Najmniejsza liczba to: -6

Process finished with exit code 0
```

3B

## Rozwiązanie 4

Wystarczy zdefiniować zmienną pomocniczą i w pętli **for** wykonać odpowiednie obliczenia, jednocześnie wyświetlając wynik.

```
poprzednia = 1
for i in range(1, 11):
    print(i + poprzednia)
    poprzednia = i
```

```
C:\Python\Projekt1\venv\Scripts\pyth
2
3
5
7
9
11
13
15
17
```

4

# 3 Programowanie w Pythonie

**W poprzednim rozdziale poznaliśmy różnego typu instrukcje proceduralne, operatory, typy podstawowe i pętle. Teraz dowiemy się, jak korzystać z tych wszystkich elementów w praktyce, i utworzymy własne funkcje na potrzeby naszych przyszłych skryptów i programów**

## Podstawowe informacje o funkcjach

Funkcja jest to w uproszczeniu zbiór instrukcji, który może być wykonywany wielokrotnie w trakcie działania programu z różnymi parametrami. Funkcje obliczają również wartość wyniku i pozwalają na określenie parametrów wejściowych. Zapisanie operacji w postaci funkcji sprawia, że staje się ona przydatnym narzędziem, z którego można korzystać w wielu różnych przypadkach w dalszej części kodu, a nawet w innych skryptach.

Najważniejsze dla nas jednak jest to, że funkcje są alternatywą dla programowania polegającego na ciągłym kopiowaniu i wklejaniu prawie tych samek linijek kodu i zmieniania parametrów wewnątrz tych linijek. Zamiast tworzyć wiele kopii tej samej serii instrukcji, możemy utworzyć jedną funkcję i wielokrotnie się do niej odwoływać. Stosowanie funkcji umożliwia później wprowadzanie zmian w samej funkcji, a nie w kilkunastu miejscach naszego kodu. Jeśli jakaś operacja może być wykonana w kodzie więcej niż jeden raz, warto dla niej utworzyć funkcję.

### WARTO WIEDZIEĆ

Funkcje są najważniejszą strukturą programu w Pythonie. Umożliwiają ponowne wykorzystanie kodu i minimalizują jego powtarzalność. Funkcje świetnie sprawdzają się w rozbudowanych projektach, ponieważ dzięki nim można rozbić skomplikowane procesy na kilka podstawowych elementów (czyli funkcji).

### Tworzenie funkcji

Do tej pory stosowaliśmy funkcje w Pythonie jedynie przy okazji różnego typu przykładów. Korzystając z funkcji **len**, uzyskaliśmy liczbę elementów danego obiektu, a funkcja **range** pozwoliła nam na definiowanie indeksowanych zakresów. Teraz dowiemy się, jak tworzyć własne nowe funkcje.

Tworzone przez nas funkcje będą zachowywały się dokładnie jak te, które są wbudowane w Pythona. Będziemy mogli je wywoływać, przekazywać parametry i uzyskiwać

z nich wyniki. Ten proces wymaga jednak poznania kilku nowych instrukcji i koncepcji – przeczytamy o nich w tym rozdziale. **Uwaga!** Jeśli programowaliśmy wcześniej w języku C lub innym kompilowanym, warto wiedzieć, że w przypadku Pythona funkcje zachowują się zupełnie inaczej.

## Koncepcje i instrukcje dotyczące funkcji:

- **Def to kod wykonywalny** – funkcje w Pythonie zapisywane są za pomocą nowej instrukcji **def**. W przeciwieństwie do funkcji z języków kompilowanych **def** jest kodem wykonywalnym – funkcja nie istnieje więc w Pythonie, dopóki Python nie dotrze do jej kodu i nie wykona instrukcji **def**. Dlatego też typowo instrukcje **def** zapisywane są w plikach modułów i wykonywane w celu wygenerowania funkcji wtedy, kiedy plik modułu zostanie zaimportowany po raz pierwszy.
- **Instrukcja def tworzy obiekt i przypisuje go do nazwy** – w momencie gdy Python dotrze do instrukcji **def** i ją wykona, generowany jest nowy obiekt funkcji, który zostaje przypisany do nazwy wskazanej jako nazwa funkcji. Co więcej możemy przechowywać dodatkowe atrybuty na potrzeby funkcji.
- **Wyrażenie lambda tworzy obiekt i zwraca go jako wynik** – funkcje można tworzyć, również korzystając z wyrażenia **lambda** – opcja ta pozwala na wykorzystanie definicji funkcji wewnątrz wiersza w miejscach, gdzie składnia instrukcji **def** nie działa.
- **Instrukcja return pozwala przesłać wynikowy obiekt do obiektu wywołującego** – w skrócie: jeśli nasza funkcja ma generować jakiś wynik, na którym nam zależy, instrukcja **return** umożliwia pobranie tego wyniku.

Z pozostałych nie będziemy korzystać i służą znacznie bardziej skomplikowanym celom. Na kolejnych stronach poruszymy natomiast ponownie temat zmiennych – globalnych oraz lokalnych – gdyż przy stosowaniu funkcji musimy szczególnie uważać przy deklarowaniu zmiennych.

## Definiowanie i wywoływanie funkcji

Wiedzę teoretyczną czas przekuć w praktykę – rozpoczniemy od zdefiniowania naszej nowej funkcji. Na potrzeby pierwszego prostego przykładu stworzymy funkcję, która będzie realizowała mnożenie.

```
#definiowanie funkcji
def mnozenie(x, y):   A
        return x * y
#wywolanie funkcji z przekazanym
wynik = mnozenie(2,4)
print(wynik)
```

```
C:\Python\Projekt1\venv\Scripts\python.exe C:/Pytho
8
```

W pierwszej części zdefiniowaliśmy funkcję o nazwie **mnozenie** A. Zawiera ona jeden wiersz instrukcji, który jest instrukcją **return** wraz z operacją arytmetyczną. Następnie w drugiej części kodu tworzymy nową zmienną, do której przypisujemy to, co zwróci nasza nowa funkcja z podanymi ręcznie parametrami. Warto już teraz zwrócić uwagę, że zadeklarowane w definicji funkcji parametry **x** oraz **y** przyjmują wartości przekazywanych parametrów przy wywołaniu.

```
#definiowanie funkcji
def mnozenie(x, y):
        return x * y
#wywolanie funkcji z przekazany
wynik = mnozenie("Test",4)
print(wynik)
```

```
C:\Python\Projekt1\venv\Scripts\python.exe C:/Pyth
TestTestTestTest   B

Process finished with exit code 0
```

Warto zwrócić uwagę na drugi wariant wywołania naszej funkcji, w tym przypadku jeden z parametrów podanych przy wywołaniu jest łańcuchem znaków. Ponieważ nie

zdefiniowaliśmy na żadnym etapie typów zmiennych, argumentów ani zwracanych wartości, otrzymaliśmy powtórzoną sekwencję **B** (na poprzedniej stronie). Już ten prosty przykład pokazuje, że jedna funkcja w Pythonie dzięki polimorfizmowi może pełnić różne cele i jej działanie jest w dużej mierze uzależnione od tego, jakie parametry przy wywołaniu przekaże użytkownik.

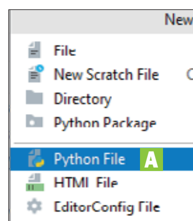
## Tworzymy moduł na nasze funkcje i z niego korzystamy

Jak widzieliśmy w przykładach, definicja funkcji jest potrzebna, zanim będziemy chcieli ją wywołać, ponieważ jeśli nie wykonamy instrukcji **def**, funkcja nie istnieje. Zamiast jednak wprowadzać definicje do różnego typu skryptów, utworzymy własny moduł na nasze funkcje, w którym będziemy je przechowywać i tworzyć, a następnie zaimportujemy go do naszego skryptu, w którym pracujemy, aby skorzystać z utworzonych funkcji. Przy importowaniu modułu wykonywane są wszystkie instrukcje **def**.

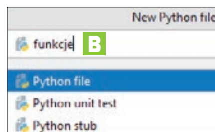
**1** W aktywnym projekcie w programie PyCharm klikamy na górnym pasku na **File**, **New**.

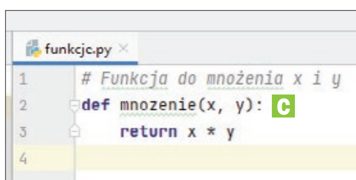
**2** Następnie klikamy na **Python File** **A**.

**3** Nadajemy nową nazwę **B** i wciskamy klawisz enter.

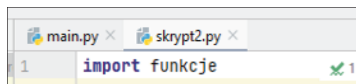



**4** Następnie do naszego nowego pliku **funkcje.py** kopiujemy definicje naszej funkcji **C** i zapisujemy zmiany. Dobrym nawykiem jest dodawanie linii komentarza z krótką informacją na temat działania funkcji.



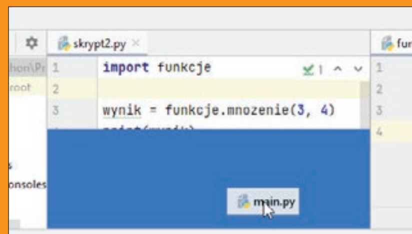

**5** Teraz w pliku naszego skryptu, gdzie będziemy tworzyć główną jego część, wpisujemy w górnej części polecenie **import [nazwa_modułu]**, w naszym przypadku **import funkcje**.

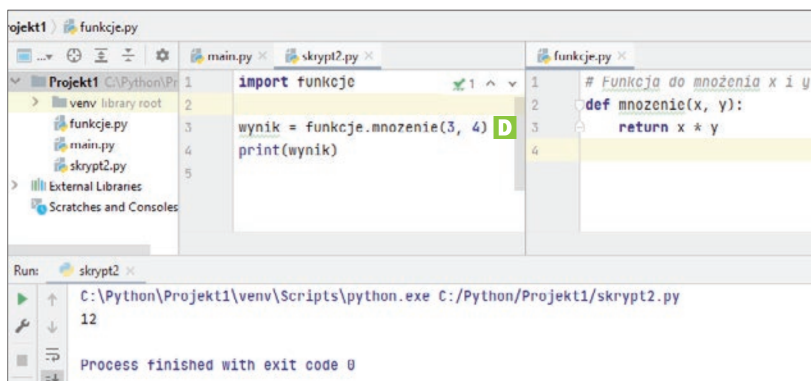



**6** Teraz możemy korzystać z funkcji z naszego modułu. Należy jednak pamiętać, że importując jedynie moduł bez jego zawartości, musimy ręcznie odwołać się do funkcji wewnątrz modułu, z której chcemy skorzystać. Składnia jest następująca: **moduł.funkcja** – u nas będzie to **funkcje.mnozenie** **D**. Oczywiście możemy również skorzystać z kilku wbudowanych funkcji, które sprawdzą się w pracy w sesji interaktywnej lub kiedy będziemy chcieli szybko uzyskać informacje na temat wszystkich funkcji, które znajdują się w danym module. Należy wpisać instrukcję **dir(nazwa_modułu)**, na przykład **dir(sys)**.

### KILKA OKIEN NA KOD

**Jeśli chcemy, aby w PyCharm główne okno, w którym wpisujemy kod, zostało podzielone na kilka stref, wystarczy kursorem przeciągnąć kartę z kodem do jednego z boków i puścić. Możemy dokonać podziału pionowego lub poziomego.**

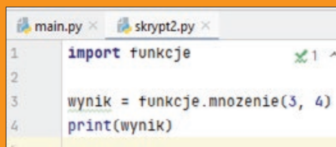




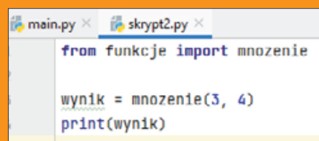

# RÓŻNE SPOSOBY IMPORTOWANIA

O importowaniu czytaliśmy już w pierwszym rozdziale. Jak widać na powyższym przykładzie, jest ono bardzo przydatne, nawet gdy nie korzystamy ze skomplikowanych modułów, ponieważ sami możemy tworzyć własne biblioteki na funkcje. Możemy dokonać importu na różne sposoby, co znacząco wpływa na sposób odwoływania się do modułu i funkcji lub zawartych w nich części kodu.

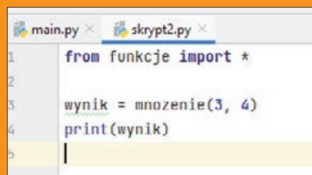
- Standardowo import wygląda następująco: wpisujemy instrukcję **import**, podajemy nazwę modułu, a następnie w kodzie odwołujemy się do samego modułu i wewnętrznych definicji.



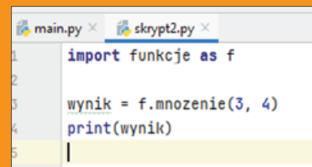

- Możemy również zaimportować bezpośrednio pojedynczą funkcję z całego modułu. Jest to dość praktyczne, gdyż czasem ładowanie ogromnego modułu w celu wykorzystania jednej czy dwóch funkcji nie jest efektywne. Dodatkowo dzięki takiemu rozwiązaniu możemy łatwiej odwoływać się do samych funkcji.




- Jest też możliwość zaimportowania wszystkich funkcji z modułu z wykorzystaniem symbolu *, nie jest to jednak zalecane, gdyż nazwy funkcji mogą się pokrywać, a my nie będziemy tego świadomi. Najlepiej odwoływać się bezpośrednio do konkretnych funkcji z konkretnych modułów.




Jeśli nazwa naszego modułu jest zbyt długa, możemy przypisać jej alias, czyli nazwę zastępczą. Należy jednak pamiętać, że w kodzie musimy później odwoływać się właśnie do aliasu.



```
import funkcje as f
import sys

print(dir(sys))
wynik = f.mnozenie(3, 4)
print(wynik)
print(dir(f))
```

```
# Funkcja do mnożenia x i y
def mnozenie(x, y):
    return x * y
```

```
C:\Python\Projekt1\venv\Scripts\python.exe C:/Python/Projekt1/skrypt2.py
['__breakpointhook__', '__displayhook__', '__doc__', '__excepthook__', '__interactivehook__', '__loader__', '__name__',
12
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', 'mnozenie']

Process finished with exit code 0
```

E

Jak widać, nasz moduł, który fizycznie ma tylko jedną definicję funkcji, automatycznie ma również przypisane podstawowe funkcje E.

## Funkcja do wyszukiwania powtórzeń w łańcuchach znaków

Utworzenie funkcji poszukującej części wspólnej dla sekwencji jest świetnym przykładem na wykorzystanie funkcji. Takie zadanie można zrealizować za pomocą pętli, jednak dla każdego łańcucha musielibyśmy odpowiednio określać zakres i modyfikować pętle. Korzystając z funkcji, możemy realizować sprawdzanie dowolnych łańcuchów, po prostu przekazując je jako parametry.

Po zdefiniowaniu nowej funkcji A i przypisaniu jej nazwy ustalamy, że funkcja będzie przyjmowała dwa argumenty, następnie tworzymy pustą listę, w której na końcu przekażemy rezultat działania funkcji. Następnie rozpoczynamy przeszukiwanie pierwszej sekwencji i jeśli dany element ciągu znaków zostanie znaleziony, również w drugiej sekwencji, korzystając z polecenia **append**, dodamy go do naszej listy wynikowej, którą zwrócimy uzyskany rezultat.

```
import funkcje as f

s1 = "Zielony"
s2 = "Niebieski"

porownaj = f.wspolna(s1, s2)
print(porownaj)
```

B

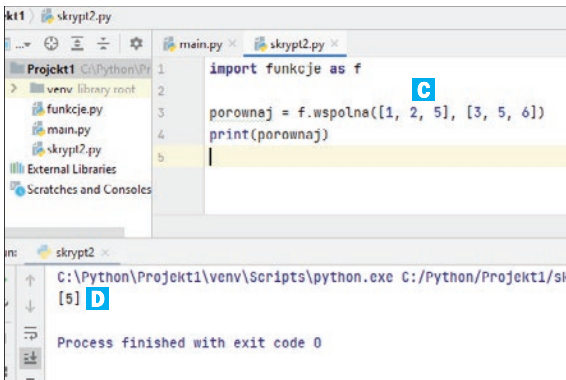
```
C:\Python\Projekt1\venv\Scripts\python.exe C:/Py
['i', 'e']
```

```
# Funkcja do znajdowania części wspólnej w sekwencjach
def wspolna(sek1, sek2):
    wynik = []
    for x in sek1:
        if x in sek2:
            wynik.append(x)
    return wynik
```

A

Teraz musimy wywołać naszą nową funkcję B. Żeby z niej skorzystać, potrzebujemy zadeklarować dwie nowe różne sekwencje, a następnie przekazać je jako argumenty przy wywołaniu funkcji.

Funkcja **wspolna** jest polimorficzna. Oznacza to, że może służyć do porównywania różnych sekwencji, nie tylko łań-




cuchów znaków. Wystarczy przekazać odpowiednie parametry **C** i ją wywołać. W tym przykładzie funkcja wskazała część wspólną z dwóch list, czyli **5** **D**.

## Zmienne lokalne i globalne

Wiemy już, czym są zmienne, jak je deklarować i przypisywać im różnego typu wartości. Teraz jednak musimy spojrzeć na nie z nieco szerszej perspektywy.

Zaczniemy od **zmiennych lokalnych**. Są to nazwy widoczne jedynie wewnątrz definicji funkcji lub wewnątrz konkretnego kawałka kodu. Istnieją jedynie w czasie wykonywania funkcji. W naszych przykładach dla funkcji **wspolna** zadeklarowaliśmy zmienną **wynik**, **sek1**, **sek2** oraz **x**. Każda z tych zmiennych jest dostępna jedynie lokalnie wewnątrz kodowanej funkcji. Nie ma możliwości odniesienia się do wartości przechowywanych przez te zmienne po zakończeniu działania funkcji. Stosowanie zmiennych lokalnych jest ogólnie zalecane. Sprawia mniej problemów na dłuższą metę i umożliwia korzystanie z tych samych często wykorzystywanych nazw dla zmiennych, na przykład **x** lub **y**.

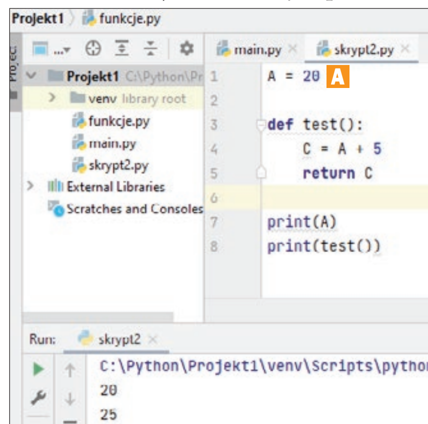
Zanim przejdziemy do dokładnego omówienia tego, czym są **zmienne globalne**, warto wiedzieć, że w Pythonie wszystkie nazwy znajdują się w **przestrzeni nazw**. Za każdym razem, gdy chcemy skorzystać z jakiejś nazwy w odniesieniu do kodu, w przestrzeni nazw do tej nazwy odnosi się konkretny zakres. To właśnie lokalizacja przypisania nazwy w kodzie określa zakres, w jakim ta nazwa będzie widoczna w kodzie. Domyślnie wszystkie nazwy przypisane wewnątrz funkcji są związane tylko i wyłącznie z jej przestrzenią nazw. Możemy przypisywać zmienne w trzech różnych miejscach:

- Wewnątrz instrukcji **def** – taka zmienna staje się zmienną lokalną dla tej funkcji.
- Wewnątrz instrukcji **def** zawierającej inną funkcję – taka zmienna staje się zmienną nielokalną dla tej funkcji.
- Zmienna przypisana poza wszystkimi instrukcjami **def** – taka zmienna staje się zmienną globalną dla całego pliku.

Zmienne globalne są więc tworzone jako ogólne wartości dla całego pliku i dostępne w jego obrębie. Tworzyliśmy już takie zmienne.

## Przykłady

Przyjrzyjmy się kilku przykładom zastosowania zmiennych lokalnych i globalnych oraz temu, jak korzystać ze zmiennych globalnych wewnątrz funkcji i ich lokalnych przestrzeni.




W tym przykładzie skorzystaliśmy ze zmiennej **A** **A** wewnątrz funkcji i uzyskaliśmy do-

stęp do jej wartości, mimo że nie zadeklarowaliśmy jej jako zmiennej wewnątrz tej funkcji. Dlatego też korzystanie ze zmiennych globalnych może sprawiać problemy, jeśli nie będziemy odpowiednio zarządzać zmiennymi. Możemy również definiować z góry, że dana zmienna będzie globalna poprzez instrukcję **global**.

```
A = 20

def test():
    A = 55        B

test()
print(A)
```

```
C:\Python\Projekt1\venv\Scripts\python.exe C:/
20  C

Process finished with exit code 0
```

Bez dodatkowych instrukcji wewnątrz funkcji próba zmiany wartości zmiennej **A** B nie przyniesie żadnego rezultatu C – w takim wypadku konieczne jest odwołanie do zmiennej globalnej.

```
A = 20

def test():
    global A      D
    A = 55

test()
print(A)
```

```
C:\Python\Projekt1\venv\Scripts\python.
55  E
```

Po dodaniu informacji, że zmienna **A** jest globalna D i tak ma być traktowana, możemy zmienić jej wartość wewnątrz funkcji E.

```
A = 20
def test():
    global A      F
    A = 55
def test2():
    global A      F
    A = 60

test2()
test()
print(A)
```

```
C:\Python\Projekt1\venv\Scripts\python.exe C:/Py
55

Process finished with exit code 0
```

Problem pojawia się w sytuacji, gdy w kilku funkcjach wykorzystujemy tę samą zmienną F – staje się ona zmienna w czasie i jej wartość jest uzależniona od tego, która funkcja zostanie wykonana po której. Dlatego też zdecydowanie nie zaleca się korzystania ze zmiennych globalnych wewnątrz funkcji – trudno jest później wyszukać błędy.

## Funkcje rekurencyjne

Rekurencja polega na wywołaniu funkcji przez samą siebie. Funkcje rekurencyjne często są alternatywą dla prostych pętli i iteracji. Zagadnienie rekurencji jest jednym z bardziej skomplikowanych – nie będziemy tu dogłębnie analizować tego zagadnienia, tylko przyjrzymy się, na czym w praktyce polega możliwość stosowania funkcji rekurencyjnych.

```
def suma(L):
    if not L:
        return 0
    else:                                   A
        return L[0] + suma(L[1:])

print(suma([1, 2, 3, 4, 5]))
```

```
C:\Python\Projekt1\venv\Scripts\python.exe C:/Python/
15
```

■ **Sumowanie z użyciem rekurencji** – przy każdym wywołaniu funkcja rekurencyjnie wywołuje samą siebie **A** w celu policzenia sumy pozostałych elementów listy, po czym ta suma jest dodawana do pierwszego odczytanego elementu. Rekurencyjne wywołania kończą się, gdy podlista jest pusta – w tym przypadku funkcja zwraca zero. Warto wiedzieć, że w tego typu wywołaniach zmienna **L** jest inną zmienną dla każdego wywołania.

■ **Obliczanie silni** – w tym przykładzie pokazane są dwa sposoby obliczania silni zrealizowane w Pythonie, pierwszy za pomocą rekurencji **B**, drugi metodą iteracji **C**. Jak widać, w przypadku silni kod z rekurencją jest znacznie krótszy i schludniejszy. Jednak praktycznie rzecz biorąc, prawie zawsze zastosowanie rekurencji jest nieopłacalne, ponieważ zużywa znacznie więcej zasobów i wymaga znacznie więcej czasu.

```
n = int(input("Prosze podac liczbe: "))

def silnia_rek(n):          B
    if n > 1:
        return n * silnia_rek(n - 1)
    elif n in (0, 1):
        return 1;
def silnia_iter(n):         C
    silnia_tmp = 1
    if n in (0, 1):
        return 1
    else:
        for i in range(2, n + 1):
            silnia_tmp = silnia_tmp * i
        return silnia_tmp

print(silnia_rek(n))
print(silnia_iter(n))
```

```
C:\Python\Projekt1\venv\Scripts\python.exe C:/Python/Projekt1
Prosze podac liczbe: 5
120
120

Process finished with exit code 0
```

# Ćwiczenia i nowe funkcje

Teraz przejdziemy do realizowania zadań, które podsumowują zdobytą przez nas wiedzę i pokażą nam, jak można wykorzystać informacje z poprzednich stron.

Utworzymy skrypty do: zgadywania liczb, odwracania słów, obliczania pola koła, usuwania samogłosek z ciągu znaków, wyszukiwania najmniejszej i największej liczby ze zbioru oraz zaprogramujemy własny kalkulator. Warto próbować napisać kod samodzielnie i dopiero w razie problemów szukać instrukcji do poszczególnych zadań na kolejnych stronach.

## Skrypt do zgadywania liczb

Do problemu zgadywania liczb możemy podejść na wiele sposobów. Najprostsza metoda to zdefiniowanie na stałe zmiennej z konkretną liczbą i utworzenie pętli, która będzie weryfikować, czy użytkownik wprowadza odpowiednią liczbę z danego mu zakresu.

Bardziej skomplikowana wersja może polegać na pseudolosowym wyborze liczby do odgadnięcia – dzięki temu przy każdym uruchomieniu programu użytkownik będzie musiał odgadnąć nową liczbę.

Można powiedzieć, że jest to pewnego rodzaju gra, którą zaprogramujemy.

W prostszej wersji naszej gry w zgadywanie liczb z góry ustalamy liczbę, którą ma odgadnąć użytkownik. Następnie definiujemy zmienną pomocniczą, która pozwoli nam na kontrolę, ile razy użytkownik może próbować zgadywać **A** (patrz strona 38). Liczba prób będzie warunkiem wyjścia z pętli **while**.

# WYDAJNOŚĆ FUNKCJI A POMIARY

Często w odniesieniu do funkcji różnego typu, na przykład sortowania, mówi się o wydajności i optymalnej pracy. Wielu początkujących programistów nie zdaje sobie sprawy, że aplikacja napisana źle pod względem optymalizacji kodu może sprawić, iż nawet wydajne komputery będą miały problem z płynną pracą. W Pythonie możemy skorzystać z narzędzi pomiarowych, które są w stanie przetestować wycinek kodu i zmierzyć czas potrzebny na wykonanie tego kodu. Skorzystamy z nich do weryfikacji czasu potrzebnego na obliczenie silni w zależności od funkcji. Możemy to zrobić na co najmniej kilka sposobów, przeanalizujemy tu trzy.

```
import timeit  A

test_czas = """
n = 900
def silnia_rek(n):
    if n > 1:
        return n * silnia_rek(n - 1)
    elif n in (0, 1):
        return 1;

a = silnia_rek(n)
"""
czas = timeit.timeit(test_czas, number=100)/100
print(czas)
```

```
C:\Python\Projekt1\venv\Scripts\python.exe C:/Python/Projekt1/skrypt2.py
0.0003192169999999999

Process finished with exit code 0
```

W celu zmierzenia czasu wykonania skryptu importujemy moduł **timeit** **A**, następnie definiujemy nową zmienną pomocniczą, której przypisujemy wartość """ i po zakończeniu naszego testowanego kodu również podajemy trzy znaki " w celu oznaczenia końca testowanego wycinka. Następnie definiujemy zmienną, która służy do wywołania funkcji badającej czas realizacji kodu. Jako argumenty podajemy nasz wycinek testowanego kodu i liczbę testów, jaką chcemy wykonać. W naszym przykładzie podajemy wartość 100, co daje dobry punkt odniesienia. Na końcu dzielimy wynik przez 100, ponieważ

```
import timeit

test_czas = """
n = 900
def silnia_iter(n):  B
    silnia_tmp = 1
    if n in (0, 1):
        return 1
    else:
        for i in range(2, n + 1):
            silnia_tmp = silnia_tmp * i
        return silnia_tmp
a = silnia_iter(n)
"""
czas = timeit.timeit(test_czas, number=100)/100
```

```
C:\Python\Projekt1\venv\Scripts\python.exe C:/Python/Projekt1/skrypt2.py
0.0002109910000000003

Process finished with exit code 0
```

w ten sposób uzyskamy uśredniony wynik jednego pomiaru dla naszego kodu. Obliczenie silni rekurencyjnie dla n = 900 zajęło średnio 0,000319 sekundy (około 3,2 x $10^{-4}$ sekundy). W przypadku funkcji iteracyjnej **B** obliczanie silni tego samego stopnia zajęło jedynie 0,000211 sekundy (2,1 x $10^{-4}$ sekundy). To sporo szybciej. Jeśli wyobrazimy sobie, że takich funkcji jest więcej lub zadanie jest bardziej skomplikowane, może okazać się, że w wypadku mniej wydajnego kodu będziemy bardzo długo czekać na realny wynik.

Sposobów na obliczanie silni jest wiele. Jednym z najkrótszych jest zastosowanie instrukcji **lambda** **C**, o której wspomnieliśmy kilka stron wcześniej. Jednak żeby z nich korzystać, trzeba być nieco bardziej zaawansowanym w kodowaniu i lepiej znać Pythona. Skorzystanie z lambdy, mimo że jest efektowne i zajmuje znacznie mniej miejsca w skrypcie, czasowo jest opłacalne podobnie jak rekurencja. Lambda z zasady jest instrukcją, która tworzy nowy obiekt, a to nie jest korzystne z punktu widzenia czasu.

Na koniec przyjrzyjmy się wynikowi dla zdefiniowanej z góry funkcji obliczania silni (**factorial**), która jest dostępna w module **math** **D**. Realizuje ona to obliczenie w czasie 2,89 x $10^{-5}$ sekundy, czyli niemal 10 razy szybciej niż nasza iteracyjna próba. Widać więc, że nie warto wymyślać niektórych implementacji samemu – lepiej korzystać z najbardziej optymalnych rozwiązań dostępnych we wbudowanych bibliotekach.

Projekt1 › skrypt2.py

```
test_czas = """
n = 900

def silnia(n):
    f = lambda x: x * f(x-1) if x != 0 else 1    C
    return f(n)

a = silnia(n)
"""
czas = timeit.timeit(test_czas, number=100)/100
print(czas)
```

```
Run: skrypt2
C:\Python\Projekt1\venv\Scripts\python.exe C:/Python/Projekt1/skrypt2.py
0.00032510500000000007
```

Projekt1 › skrypt2.py

```
import timeit
ust = "from math import factorial"    D

test_czas = """
n = 900

a = factorial(n)
"""
czas = timeit.timeit(setup=ust, stmt=test_czas, number=100)/100
print(czas)
```

```
Run: skrypt2
C:\Python\Projekt1\venv\Scripts\python.exe C:/Python/Projekt1/skrypt2.py
2.8916000000000012e-05

Process finished with exit code 0
```

```
zgadnij = 7
print(zgadnij)

ile_prob = 1
while ile_prob < 7:
    liczba = int(input("Podaj liczbę: "))

    if liczba == zgadnij:
        print("Zgadłeś!")
        break
    elif liczba < zgadnij:
        print("Zbyt niska wartość")
    else:
        print("Zbyt wysoka wartość")

    ile_prob += 1

print("Zgadłeś za :", ile_prob, "razem")
```

A

```
C:\Python\Projekt1\venv\Scripts\python.exe C:/Python/Projekt1/skrypt2.py
7
Podaj liczbę: 3
Zbyt niska wartość
Podaj liczbę: 22
Zbyt wysoka wartość
Podaj liczbę: 7
Zgadłeś!
Zgadłeś za : 3 razem
```

B

Wewnątrz pętli pobieramy liczbę od użytkownika. Następnie tworzymy instrukcję warunkową, w której sprawdzamy, czy liczba wprowadzona jest równa liczbie, która została zdefiniowana na początku. Jeśli tak, to wyświetlamy odpowiednią informację B i opuszczamy pętlę, korzystając z instrukcji **break**. W innym przypadku wyświetlamy jedną z dwóch pozostałych możliwości. W celu zwiększenia trudności zastąpimy z góry zdefiniowaną liczbę przez generowaną losowo (tak naprawdę pseudolosowo). W tym celu w Pythonie musimy skorzystać z modułu **random**, a następnie funkcji dostępnej w tym module **randrange**, który przyjmuje dwa parametry, dolny i górny zakres, z jakiego chcemy uzyskać liczbę. Uzyskane liczby domyślnie są całkowite. Pozostały kod programu pozostaje bez zmian.

## LOSOWE A PSEUDOLOSOWE

Liczba losowa jest generowana w trakcie działania określonego z góry mechanizmu losującego, którego każdy element jest jasno określony. Przykładem może być liczba oczek na rzuconej przez nas kości do gry. W przypadku komputerów uzyskanie liczb losowych jest bardzo trudne, dlatego wprowadzono specjalne algorytmy, które pozwalają na uzyskanie liczb pseudolosowych, czyli takich, których mechanizm uzyskiwania ma ukryte regularności, co można teoretycznie wykorzystać, ale są one nieistotne z punktu widzenia zwykłego użytkownika.

Oczywiście, jeśli chcemy naprawdę zagrać w zgadywanie, musimy wykomentować lub usunąć wiersz, który odpowiada za wyświetlanie wylosowanej liczby. Jest on dodany ze względu na to, że pozwala sprawdzić, czy losowanie przebiega poprawnie.

```
import random

zgadnij = random.randrange(1, 201)

print(zgadnij)

ile_prob = 1
while ile_prob < 10:
    liczba = int(input("Podaj liczbę: "))

    if liczba == zgadnij:
        print("Zgadłeś!")
        break
    elif liczba < zgadnij:
        print("Zbyt niska wartość")
    else:
        print("Zbyt wysoka wartość")

    ile_prob += 1

print("Koniec gry, zgadywałeś:", ile_prob, "razy")
```

## Skrypt do odwrócenia ciągów znaków

Dość często stawia się programistów przed zadaniem napisania programu, który będzie odwracał ciąg znaków. W większości przypadków takie zadanie da się rozwiązać na kilka sposobów.
W Pythonie wystarczy napisać zaledwie kilka linijek kodu.
Zaczniemy od testowego kodu **A**, który będzie służył jedynie do sprawdzenia, czy tworzona przez nas funkcja zadziała prawidłowo. Deklarujemy zmienną i przypisujemy do niej ciąg znaków. Następnie definiujemy funkcję, której zadaniem będzie odwracanie tekstu. Polecenie **print-(string[::-1])** realizuje całe nasze zadanie.
Aby całe zadanie rozwiązać w kilku linijkach, przydaje się dobra znajomość zapisu wycinków – **list[<start>:<stop>:<krok>]** – jak widać, zapis **[::-1]** oznacza, że zaczniemy „odcinać” od końca do początku co jeden znak i w ten sposób uzyskamy odwrócenie ciągu wejściowego.
Aby nasz kod był bardziej uniwersalny, wystarczy zamienić sztywne przypisanie łańcucha znaków – na prośbę o wpisanie dowolnego łańcucha przez użytkownika **B**.
Po zatwierdzeniu takiego łańcucha klawiszem enter od razu pojawi się wynik.

```
tekst = "testowy kawałek tekstu"

def odwracanie(string):
    print(string[::-1])

odwracanie(tekst)
```

```
C:\Python\Projekt1\venv\Scripts\python.exe C:/Python/
utsket keławak ywotset

Process finished with exit code 0
```

A

```
tekst = input("Proszę podać tekst do odwrócenia: ")

def odwracanie(string):
    print(string[::-1])

odwracanie(tekst)
```

```
C:\Python\Projekt1\venv\Scripts\python.exe C:/Python/Projekt1/skrypt2.py
Proszę podać tekst do odwrócenia: Komputer Świat!
!taiwŚ retupmoK

Process finished with exit code 0
```

B

## Skrypt do obliczania pola koła

Teraz zajmiemy się prostym problemem obliczania powierzchni koła.
Wzór pewnie wszyscy pamiętają, a jeśli nie, to można go w chwilę znaleźć w sieci. Problemem może się okazać implementacja stałej pi w naszych obliczeniach. Najprościej byłoby przyjąć, że ma ona wartość 3,14, i na tej podstawie dalej liczyć. Bardziej profesjonalnie jest jednak skorzystać z wartości pi umieszczonej w module **math**. Dodatkowo w tym przykładzie skorzystamy również z nieskończonej pętli **while** z odpowiednim warunkiem wyjścia, co pozwoli nam wygodnie obliczać kolejne pola powierzchni bez konieczności ponownego uruchamiania skryptu.
W trakcie tworzenia tego skryptu po raz pierwszy trafimy na problem, który może powodować dynamiczne typowanie. Do tej pory nie musieliśmy się martwić tym, jaki typ przyjmie zmienna, z której korzystaliśmy. W przypadku operacji mnożenia jednak pojawia się kłopot. W trakcie obliczania zmiennej **pole** wykonujemy mnożenie liczb, a tymczasem domyślnie przy wprowadzeniu przez użytkownika zmiennej **promien** została ona zakwalifikowana jako **string (typ tekstowy)** **A** (patrz kolejna strona). Ponieważ Python nie zezwala na wykonywanie operacji mnożenia typów **float (zmiennoprzecinkowe)** z typami **non-int**, mamy błąd krytyczny **B**. Możemy po-

```
# Skrypt do obliczania pola powierzchni koła
import math

print("Gdy chcesz zakończyc prace programu wprowadź X jako promień")
while True:
    promien = input("Wprowadz promień okręgu: ")
    if promien == "x":  A
        break
    else:
        pole = math.pi*promien*promien
        print("Pole powierzchni wynosi: ", pole)
```

```
C:\Python\Projekt1\venv\Scripts\python.exe C:/Python/Projekt1/skrypt2.py
Gdy chcesz zakończyc prace programu wprowadź X jako promień
Wprowadz promień okręgu: 5
Traceback (most recent call last):
  File "C:\Python\Projekt1\skrypt2.py", line 10, in <module>
    pole = math.pi*promien*promien
TypeError: can't multiply sequence by non-int of type 'float'  B

Process finished with exit code 1
```

```
else:
    pole = math.pi*promien*promien
    print("Pole powierzchni wynosi: ", pole)
```

Variables
- __exception__ = {tuple: 3} (<class 'TypeError'>, TypeError("...
- promien = {str} '5'
- Special Variables

C

dejrzeć, jaki typ jest przypisywany do jakiej zmiennej, korzystając z **debuggera** C – więcej o tym narzędziu przeczytamy w kolejnych rozdziałach.

Możemy naprawić ten problem poprzez konwersję. Wystarczy, że na sztywno przypiszemy typ do zmiennej. Dlatego też dodajemy nową zmienną D, która przyjmie wartość wprowadzoną przez użytkownika i przechowa ją jako typ **float**. Umożliwi to wykonywanie obliczeń.

Tak więc przy pobieraniu danych od użytkownika należy pamiętać o tym, jakiego typu dane chcemy przetwarzać i jakiego typu wyniku oczekujemy.

## LICZBY ZMIENNOPRZECINKOWE – PRZECINKI I KROPKI

W polskiej notacji przyjęło się, że liczby z częścią ułamkową zapisuje się z przecinkiem między częścią całkowitą a ułamkową, na przykład 3,67. W angielskiej notacji jednak zamiast przecinka stosuje się kropkę. Python jest angielskojęzyczny, dlatego pamiętajmy, aby zawsze wprowadzać liczby zmiennoprzecinkowe w notacji angielskiej – w innym wypadku pojawi się błąd.

```
C:\Python\Projekt1\venv\Scripts\python.exe C:/Python/Projekt1/skrypt2.py
Gdy chcesz zakończyc prace programu wprowadź X jako promień
Wprowadz promień okręgu: 3.6
Pole powierzchni wynosi:  40.71504079052372
Wprowadz promień okręgu: 3,6
Traceback (most recent call last):
  File "C:\Python\Projekt1\skrypt2.py", line 15, in <module>
    policz_pole_okr(promien)
  File "C:\Python\Projekt1\skrypt2.py", line 8, in policz_pole_okr
    promien2 = float(promien)
ValueError: could not convert string to float: '3,6'

Process finished with exit code 1
```

```
# Skrypt do obliczania pola powierzchni koła
import math

print("Gdy chcesz zakończyc prace programu wprowadź X jako promień")
while True:
    promien = input("Wprowadz promień okręgu: ")
    if promien == "x":
        break
    else:
        promien2 = float(promien)  D
        pole = math.pi*promien2*promien2
        print("Pole powierzchni wynosi: ", pole)
```

```
C:\Python\Projekt1\venv\Scripts\python.exe C:/Python/Projekt1/skrypt2.py
Gdy chcesz zakończyc prace programu wprowadź X jako promień
Wprowadz promień okręgu: 5
Pole powierzchni wynosi:  78.53981633974483
Wprowadz promień okręgu: x

Process finished with exit code 0
```

Możemy również zapakować obliczanie pola powierzchni w funkcję i zakończyć działanie programu, korzystając z instrukcji **exit()** **E**, z której jeszcze nie korzystaliśmy. Powoduje ona zakończenie działania aktywnego skryptu. Korzystanie z funkcji znacząco skraca kod samego programu i upraszcza go, gdy wszystkie zbędne obliczenia są realizowane poza głównym kodem skryptu.

```
# Skrypt do obliczania pola powierzchni koła
import math

def policz_pole_okr(promien):
    if promien == "x":
        exit()  E
    else:
        promien2 = float(promien)
        pole = math.pi*promien2*promien2
        print("Pole powierzchni wynosi: ", pole)

print("Gdy chcesz zakończyc prace programu wprowadź X jako promień")
while True:
    promien = input("Wprowadz promień okręgu: ")
    policz_pole_okr(promien)
```

```
C:\Python\Projekt1\venv\Scripts\python.exe C:/Python/Projekt1/skrypt2.py
Gdy chcesz zakończyc prace programu wprowadź X jako promień
Wprowadz promień okręgu: 3.8
Pole powierzchni wynosi:  45.36459791783661
Wprowadz promień okręgu: 6
Pole powierzchni wynosi:  113.09733552923255
Wprowadz promień okręgu: x

Process finished with exit code 0
```

## Skrypt do usuwania samogłosek lub spółgłosek z tekstu

W celu rozwiązania tego zadania możemy stworzyć albo dwa osobne proste programy – jeden będzie usuwał samogłoski, a drugi spółgłoski – albo jeden złożony, którego działanie będzie uzależnione od wyboru użytkownika. Problem nie jest skomplikowany. Należy utworzyć listy spółgłosek i samogłosek, które mają być usuwane z wprowadzanego tekstu. Następnie musimy przeskanować cały wprowadzony tekst i po znalezieniu litery pasującej do wzorca wstawić w jej miejsce pustą przestrzeń. Jak wiemy, łańcuchy tekstowe są obiektami niezmiennymi, więc w celu usunięcia znaku musimy wygenerować nowy obiekt, który nie będzie go zawierał.

W tym przypadku również skorzystamy z nieskończonej pętli, z której możemy wyjść, wprowadzając znak **x** jak zmienną tekstową **A**. Następnie musimy utworzyć nową zmienną pomocniczą na tekst, z którego będą usuwane po kolei znaki. Wewnątrz pętli **for** skorzystaliśmy z funkcji wbudowanej łańcuchów znaków **lower** **B**, pozwala nam to na przekształcenie całego ciągu na małe litery. Jest to konieczne, ponieważ użytkownik może wprowadzać dowolnie duże lub małe litery, a my definiujemy listę samogłosek małymi literami i z nimi właśnie będziemy dokonywać porównania. Następnie w celu usunięcia znaku i zastąpienia go pustym miejscem od nowa tworzymy obiekt dla nowego tekstu i korzystamy z wbudowanej funkcji **replace**, przyjmuje ona dwa argumenty, pierwszy to znak, który ma być zastąpiony, a drugi to wartość, czym ma być zastąpiony.

Jeśli chcemy utworzyć skrypt, który ma usuwać spółgłoski **C**, wystarczy podmienić kilka wartości w poprzednim przykładzie.

Nam jednak zależy na przepisaniu operacji zamiany znaków na osobne funkcje i daniu użytkownikowi możliwości wyboru **D**, jakie znaki dla jakiego ciągu mają być usuwane. Dla uproszczenia zadania zakładamy, że każdy znak niebędący samogłoską jest spółgłoską. Na początku skryptu tworzymy więc dwie funkcje, które zasadniczo różnią

```
# Skrypt do usuwania z tekstu samogłosek

print("Gdy chcesz zakończyc pracę programu wprowadź X")
while True:
    tekst = input("Wprowadz tekst: ")
    if tekst == "x":  A
        exit()
    else:
        nowy_tekst = tekst
        print("Usuwamy samogłoski z tekstu")
        samogloski = ["i", "y", "e", "a", "o", "u", "ą", "ę"]
        for i in tekst.lower():  B
            if i in samogloski:
                nowy_tekst = nowy_tekst.replace(i,"")
        print("Nowy tekst po usunięciu samogłosek: ", nowy_tekst)
```

```
C:\Python\Projekt1\venv\Scripts\python.exe C:/Python/Projekt1/skrypt2.py
Gdy chcesz zakończyc pracę programu wprowadź X
Wprowadz tekst: Boisko
Usuwamy samogłoski z tekstu
Nowy tekst po usunięciu samogłosek:  Bsk
Wprowadz tekst: x

Process finished with exit code 0
```

się jedynie warunkiem testu logicznego, a następnie dodajemy w głównej pętli **while** alternatywne testy logiczne, które pozwolą skorzystać użytkownikowi z odpowiedniej opcji programu.

```
Code  Refactor  Run  Tools  VCS  Window  Help    Projekt1 - skrypt2.py

main.py   skrypt2.py   zgadnij_liczbe.py   math.py

# Skrypt do usuwania z tekstu samogłosek lub spółgłosek   C

def usun_samogloski(tekst):
    samogloski = ["i", "y", "e", "a", "o", "u", "ą", "ę"]
    nowy_tekst = tekst
    for i in tekst.lower():
        if i in samogloski:
            nowy_tekst = nowy_tekst.replace(i, "")
    return nowy_tekst

def usun_spolgloski(tekst):
    samogloski = ["i", "y", "e", "a", "o", "u", "ą", "ę"]
    nowy_tekst = tekst
    for i in tekst.lower():
        if i not in samogloski:
            nowy_tekst = nowy_tekst.replace(i, "")
    return nowy_tekst

print("Gdy chcesz zakończyc pracę programu wprowadź x")
while True:
    wybor = input("Chcesz usunąć samogłoski wpisz 1, "
                  "chcesz usunąć samogłoski wpisz 2: ")
    tekst = input("Wprowadź tekst: ")
    if tekst == "x":
        exit()
    elif wybor == "1":
        print("Usuwamy samogłoski z tekstu")
        nowy_tekst = usun_samogloski(tekst)
        print("Nowy tekst po usunięciu samogłosek: ", nowy_tekst)
    elif wybor == "2":
        print("Usuwamy spółgłoski z tekstu")
        nowy_tekst = usun_spolgloski(tekst)
        print("Nowy tekst po usunięciu spółgłosek: ", nowy_tekst)
    else:
        print("Wprowadzono błędne dane wejściowe")
```

## Skrypt do wyszukiwania największej i najmniejszej liczby ze zbioru

Wyszukiwanie maksimum i minimum jest często bardzo przydatne. Najczęściej odnosi się do liczb, jednak czasem możemy również spotkać się z zadaniem wymagającym sprawdzenia wielkości pliku lub liczby znaków w ciągu tekstowym. Najpierw stworzymy skrypt służący do wyszukiwania minimum. Zakładamy, że funkcja, którą tworzymy, ma pozwolić na znalezienie wartości minimalnej dla dowolnego zbioru argumentów i dowolnego zbioru typów danych obiektów. Należy więc przyjąć, że funkcja ta powinna przyjmować zero lub większą liczbę argumentów – właściwie dowolną liczbę, jaką będziemy chcieli. Co więcej, funkcja ta powinna działać na wszystkich rodzajach obiektów Pythona: liczbach, łańcuchach znaków, listach, plikach itp.

```
C:\Python\Projekt1\venv\Scripts\python.exe C:/Python/Projekt1/skrypt2.py
Gdy chcesz zakończyc pracę programu wprowadź x
Chcesz usunąć samogłoski wpisz 1, chcesz usunąć samogłoski wpisz 2: 2   D
Wprowadź tekst: test
Usuwamy spółgłoski z tekstu
Nowy tekst po usunięciu spółgłosek:  e
Chcesz usunąć samogłoski wpisz 1, chcesz usunąć samogłoski wpisz 2: 1
Wprowadź tekst: testowe
Usuwamy samogłoski z tekstu
Nowy tekst po usunięciu samogłosek:  tstw
Chcesz usunąć samogłoski wpisz 1, chcesz usunąć samogłoski wpisz 2: ddst
Wprowadź tekst: dsgf
Wprowadzono błędne dane wejściowe
Chcesz usunąć samogłoski wpisz 1, chcesz usunąć samogłoski wpisz 2: x
Wprowadź tekst: x

Process finished with exit code 0
```

A

```
# Skrypt do znajdowania minimum lub maksimum

def min1(*args):
    wynik = args[0]
    for arg in args[1:]:
        if arg < wynik:
            wynik = arg
    return wynik

print(min1(4,8,33,2))
```

```
C:\Python\Projekt1\venv\Scripts\python.exe C:/Python/Projekt1/skr
2
```

Opis założeń przeważnie jest znacznie dłuższy niż sam kod funkcji. Warto też pamiętać, że każdą funkcję można zrealizować na kilka sposobów.

W naszym wypadku pierwszy sposób **A** polega na pobraniu pierwszego argumentu i iteracji przez pozostałe. Odcinamy pierwszy argument, bo nie ma sensu porównywać go z samym sobą, i jeśli porównywany obiekt jest mniejszy, zostaje przypisany do zmiennej zwracającej wynik na końcu funkcji.

B

```
# Skrypt do znajdowania minimum lu

def min2(pierwszy,*args):
    for arg in args:
        if arg < pierwszy:
            pierwszy = arg
    return pierwszy

print(min2("y","c","z"))
```

```
C:\Python\Projekt1\venv\Scripts\python.exe C:/Pytho
c

Process finished with exit code 0
```

Drugi sposób rozwiązania **B** jest nieco krótszy, ponieważ wyeliminowaliśmy problem porównywania pierwszego elementu z samym sobą poprzez utworzenie osobnego argumentu, który przechowuje wartość pierwszego elementu na początku działania funkcji. W trakcie jej działania do tej zmiennej przypisywane są najmniejsze wartości i na końcu zwracana jest jej wartość jako wynik.

Trzecia metoda rozwiązania **C** polega na konwersji krotki, w której znajdują się nasze argumenty, na listę, a następnie na sortowaniu całej listy za pomocą wbudowanej metody **sort**. Domyślnie metoda **sort** umieszcza najmniejszy element na pierwszym miejscu listy, a największy na jej końcu. W celu odczytania wyniku wystarczy więc odwołać się do odpowiedniego indeksu.

C

```
# Skrypt do znajdowania minimum

def min3(*args):
    tmp = list(args)
    tmp.sort()
    return tmp[0]

print(min3([2,2],[5,6],[1,3]))
```

```
C:\Python\Projekt1\venv\Scripts\python.exe C:/Pyt
[1, 3]
```

Ostatnie rozwiązanie jest najprostsze, jednak nie zawsze najbardziej wydajne. Można

## FUNKCJE PRZYJMUJĄCE DOWOLNĄ ILOŚĆ ARGUMENTÓW

Do zrealizowania opisanego powyżej skryptu potrzebujemy funkcji, która akceptuje dowolną ilość argumentów. Python pozwala na zdefiniowanie takiej funkcji poprzez skorzystanie z opcji *, na przykład **def funkcja(*args)**. Taki zapis oznacza, że wewnątrz funkcji możemy rozpakować listę argumentów i traktować ją jak **krotkę**, czyli niezmienną listę.

Projekt1 › skrypt2.py

main.py | skrypt2.py | zgadnij_liczbe.py | math.py

Projekt1 C:\Python\
- venv library root
- funkcje.py
- main.py
- odwracanie_ciagu
- pole_kola.py
- skrypt2.py
- usuwanie_samog
- zgadnij_liczbe.py
- External Libraries
- Scratches and Conso

```
# Skrypt do znajdowania minimum lub maksimu

def min1(*args):
    wynik = args[0]
    for arg in args[1:]:
        if arg < wynik:
            wynik = arg
    return wynik
def min2(pierwszy,*args):
    for arg in args:
        if arg < pierwszy:
            pierwszy = arg
    return pierwszy
def min3(*args):
    tmp = list(args)
    tmp.sort()
    return tmp[0]

print(min1(4,8,33,2,76))
print(min2(4,2,33,4,76))
print(min3(4,8,2,33,76))
```

Run: skrypt2

```
C:\Python\Projekt1\venv\Scripts\python.exe C:/Python/Projekt1
2
2
2

Process finished with exit code 0
```

Wszystkie funkcje zwracają minimum z dowolnego zbioru

przygotować sobie ogromne zbiory danych, w których będzie wyszukiwane minimum, i sprawdzić, która funkcja jest najbardziej optymalna. Rezultaty na pewno będą ciekawe, gdyż w zależności od zestawu danych i ich typu różne funkcje okazują się najlepsze. Jeśli więc zależy nam na uzyskaniu maksimum w zależności od przykładowej funkcji, należy jedynie zmienić znak dla porównania oraz nazwę funkcji **D**. Jeśli zamierzamy skorzystać z prostego rozwiązania problemu znajdowania minimum i maksimum, warto również skorzystać z możliwości instrukcji **lambda**.

**D**

```
def max1(*args):
    wynik = args[0]
    for arg in args[1:]:
        if arg > wynik:
            wynik = arg
    return wynik

print(max1(3,9,23,4,1))
```

**E**

```
def minmax(test,*args):
    wynik = args[0]
    for arg in args[1:]:
        if test(arg, wynik):
            wynik = arg
    return wynik

def mniejsze(x, y):return x < y
def wieksze(x, y):return x > y

print(minmax(mniejsze,3,9,23,4,1))
print(minmax(wieksze,3,9,23,4,1))
```

Run: skrypt2

```
/home/krzysztof_dziedzic,
1
23

Process finished with ex
```

Tworzymy funkcję, która pozwoli na sprawdzanie warunku logicznego w zależności od funkcji wewnętrznej **E**. Nasza funkcja przyjmuje minimum jeden argument, którym będzie funkcja **lambda** z testem logicznym. Kolejne argumenty będą dowolnie obszerną listą danych zapisanych w postaci krotki. Kluczowe jest wywołanie wewnętrzne jednej z funkcji lambda jako argumentu **test**. Dzięki temu w jednej funkcji bez dodatkowych modyfikacji możemy wywoływać dowolne funkcje porównawcze.

## Tworzymy własny kalkulator

Wykorzystując wszystkie zdobyte do tej pory informacje, możemy stworzyć nieco bardziej skomplikowany skrypt. W naszym przykładzie będzie to kalkulator. Dość prosty – taki, który będzie pozwalał na wykonywanie czterech operacji: dodawania, odejmowania, mnożenia i dzielenia. Użytkownik będzie

## WARTO WIEDZIEĆ

Tworzony przez nas skrypt miał na celu pokazanie pewnego wzorca programowania, w którym tworzymy funkcję, która może przyjąć dowolną ilość argumentów, oraz funkcję, która wywołuje kolejną funkcję wewnątrz siebie. Jeśli jednak chcemy w sposób wydajny odnaleźć maksimum i minimum, warto skorzystać z wbudowanych w Pythona funkcji – **min** oraz **max**. Funkcje te zostały zoptymalizowane pod względem szybkości wykonywania i napisane w języku C. Wystarczy jako argument przekazać zbiór danych, z których chcemy uzyskać maksimum lub minimum.

main.py | skrypt2.py | zgadnij_liczbe.py

```
print(min(4,8,33,2,76))
print(max(4,2,33,4,76))
```

Run: skrypt2

```
C:\Python\Projekt1\venv\Scripts\pyt
2
76

Process finished with exit code 0
```

wprowadzał dwie liczby, które będą poddawane jednej z operacji matematycznych, a następnie będzie mógł wybrać operację kalkulatora lub zakończyć jego działanie. Zaczynamy od opisu działania programu – będzie on wyświetlany użytkownikowi przy uruchomieniu skryptu **A**.

Główna część naszego kalkulatora będzie wykonywana w pętli **while**. Warto zwrócić uwagę, że przed instrukcjami **input** deklarujemy typ **int** – domyślnie wartość instrukcji **input** ma przypisany typ **string**, my chcemy jednak wykonywać operacje na liczbach całkowitych, stąd przypisanie typu **int**.

W dalszej części pętli **B**, korzystając z zagnieżdżenia warunków logicznych, tworzymy scenariusz dla każdej z opcji, jakie dajemy użytkownikowi.

Jeśli użytkownik nie wpisze żadnej z akceptowanych opcji, pętla zostanie powtórzona z informacją o błędzie.

Każda z naszych matematycznych operacji jest przetwarzana przez funkcje. Pozwala to uniknąć skom-

main.py | skrypt2.py | zgadnij_liczbe.py | math.py

**A**

```
# Prosty kalkulator

print("1, Dodawanie")
print("2, Odejmowanie")
print("3, Mnożenie")
print("4, Dzielenie")
print("5, Wyjście")

while True:
    wybor = int(input("Podaj numer opcji: "))
    if (wybor >= 1 and wybor <= 4):
        print("Wprowadz dwie liczby")
        num1 = int(input("Wprowadź pierwszą liczbę: "))
        num2 = int(input("Wprowadź drugą liczbę: "))
```

**B**

```
while True:
    wybor = int(input("Podaj numer opcji: "))
    if (wybor >= 1 and wybor <= 4):
        print("Wprowadz dwie liczby")
        num1 = int(input("Wprowadź pierwszą liczbę: "))
        num2 = int(input("Wprowadź drugą liczbę: "))
        if wybor == 1:
            dodaj(num1,num2)
        elif wybor == 2:
            odejmij(num1,num2)
        elif wybor == 3:
            mnozenie(num1,num2)
        else:
            dziel(num1,num2)
    elif wybor == 5:
        break
    else:
        print("Wprowadź poprawny numer opcji")
```

plikowania kodu w samej pętli. Funkcje deklarujemy w osobnym module i importujemy go lub u góry naszego skryptu.
Funkcje odpowiadające za dodawanie, odejmowanie, mnożenie i dzielenie są bardzo proste, warto jednak, nawet jeśli są długości jednego wiersza, tworzyć dla nich funkcje, gdyż w przyszłości możemy je wykorzystać na różne inne sposoby.

```
# Prosty kalkulator

def dodaj(x,y):
    print("Wynik to: ",x + y)

def odejmij(x,y):
    print("Wynik to: ",x - y)

def mnozenie(x,y):
    print("Wynik to: ",x * y)

def dziel(x,y):
    print("Wynik to: ",x / y)
```

Działanie naszego kalkulatora możemy przetestować bezpośrednio w PyCharm. Warto samemu sprawdzić, jaki będzie rezultat dzielenia dwóch liczb całkowitych, dla których wynik nie będzie liczbą całkowitą.

```
C:\Python\Projekt1\venv\Scripts\
1, Dodawanie
2, Odejmowanie
3, Mnożenie
4, Dzielenie
5, Wyjście
Podaj numer opcji: 1
Wprowadz dwie liczby
Wprowadź pierwszą liczbę: 1
Wprowadź drugą liczbę: 2
Wynik to:  3
Podaj numer opcji: 2
Wprowadz dwie liczby
Wprowadź pierwszą liczbę: 1
Wprowadź drugą liczbę: 2
Wynik to:  -1
```

## Rozbudowa kalkulatora

Możemy w każdej chwili rozbudować nasz kalkulator, dodając do niego kolejne funkcje. W celu wykonania takiej rozbudowy musimy jednak pamiętać o zachowaniu spójności całego skryptu.

**1** Musimy rozbudować kod, który jest wyświetlany użytkownikowi.

```
print("2, Odejmowanie")
print("3, Mnożenie")
print("4, Dzielenie")
print("5, Pierwiastek kwadratowy")
print("6, Wyjście")
```

**2** Następnie dodajemy kolejną instrukcję warunkową dla naszej nowej opcji.

```
    elif wybor == 4:
        dziel(num1,num2)
    else:
        pierwiastek(num1,num2)
elif wybor == 6:
    break
else:
```

**3** Definiujemy nową funkcję – w przypadku pierwiastka musimy zaimportować jeszcze moduł **math**, aby móc skorzystać z funkcji **sqrt**.

```
def pierwiastek(x,y):
    print("Wynik dla pierwszej liczby: ", math.sqrt(x))
    print("Wynik dla drugiej liczby: ", math.sqrt(y))
```

**4** Teraz nasz kalkulator został rozbudowany o jedną nową funkcjonalność.

```
5, Pierwiastek kwadratowy
6, Wyjście
Podaj numer opcji: 5
Wprowadz dwie liczby
Wprowadź pierwszą liczbę: 2
Wprowadź drugą liczbę: 4
Wynik dla pierwszej liczby:  1.4142135623730951
Wynik dla drugiej liczby:  2.0
Podaj numer opcji:
```

# 4 Obsługa plików i czasu w Pythonie

**W tym rozdziale poznamy obsługę plików w Pythonie, a także nauczymy się pracować z przetwarzaniem czasu oraz ogólnie z datami**

## Pliki

Każdy wie, czym są pliki i że mogą mieć różne rozszerzenia. Dla nas najwygodniejsza jest definicja: pliki to pojemniki na dane, którymi zarządza system operacyjny, jest to również typ wbudowany w Pythona, którym można zarządzać z poziomu skryptów. W dużym skrócie: wbudowana funkcja **open** tworzy plik obiektu Pythona, który służy jako łącze do pliku znajdującego się na komputerze. Po jej wywołaniu będziemy mogli przenieść łańcuchy znaków danych do powiązanego pliku zewnętrznego oraz z niego, wywołując metody obiektu zwracanego pliku. Oznacza to, że na nasze potrzeby będziemy mogli odczytywać i zapisywać dane w plikach.
Gdy porównamy pliki do innych typów, jakie poznaliśmy, okazują się one dość unikalne. Nie są traktowane jak liczby czy też sekwen-

### POPULARNE OPERACJE NA PLIKACH

| OPERACJA | OPIS |
|---|---|
| output = open(r'C:\plik', 'w') | Utworzenie pliku do zapisu |
| input = open('data', 'r') | Utworzenie pliku do odczytu |
| input = open('data') | To samo co powyżej – odczyt jest trybem domyślnym |
| aString = input.read() | Wczytanie całego pliku do jednego łańcucha znaków |
| aString = input.read(N) | Wczytanie następnych N bajtów do łańcucha znaków |
| aString = input.readline() | Wczytanie następnego wiersza do łańcucha znaków |
| aList = input.readlines() | Wczytanie całego pliku do listy łańcuchów znaków z poszczególnymi wierszami |
| output.write(aString) | Zapisanie łańcucha bajtów do pliku |
| output.writelines(aList) | Zapisanie do pliku wszystkich łańcuchów znaków z wierszami, które znajdują się na liście |
| output.close() | Ręczne zamknięcie pliku |
| output.flush() | Opróżnienie bufora wyjściowego bez zamykania pliku |
| anyFile.seek(N) | Zmiana pozycji w pliku na wartość przesunięcia N dla następnej operacji |

cje, nie reagują na operatory wyrażeń. Zamiast tego wszystkiego eksportują metody służące do wykonywania różnych popularnych zadań związanych z przetwarzaniem plików. Zdecydowana większość metod odpowiada za pobieranie danych wejściowych z plików zewnętrznych i zapisywanie do nich danych wyjściowych. Istnieją również metody, które umożliwiają odszukanie pozycji w pliku lub opróżnienie bufora wyjściowego. Zanim zaczniemy korzystać z plików, musimy poznać moduł **os**, który pozwoli nam poruszać się po systemie Windows i jego strukturze plików oraz folderów. Dzięki temu będziemy mogli z większą świadomością tworzyć nowe pliki, uzyskiwać do nich dostęp i sprawdzać ich lokalizację.

## Poruszamy się po systemie z modułem os

Moduł **os** jest w Pythonie modułem wbudowanym i umożliwia dostęp do funkcji, które pozwalają na poruszanie się po systemie Windows w sposób podobny jak w Wierszu polecenia. Zobaczmy kilka przykładów w interaktywnej sesji IDLE.

Zaczynamy standardowo od importu modułu. W tym wypadku importujemy sam moduł i do konkretnych funkcji będziemy się odnosić z jego uwzględnieniem, aby kod był czytelniejszy.

```
IDLE Shell 3.9.5
File  Edit  Shell  Debug
Python 3.9.5 (tags/
D64)] on win32
Type "help", "copyr
>>> import os
>>> |
```

```
Type "help", "copyright", "credits" or "license()" for more information.
>>> import os
>>> os.getcwd() A
'C:\\Users\\krzys\\AppData\\Local\\Programs\\Python\\Python39'
>>>
```

**A os.getcwd()** – ta funkcja zwraca nam ścieżkę, w której aktualnie pracujemy. Należy również zwrócić uwagę na to, że pomiędzy kolejnymi folderami znajdują się dwa znaki \\, a nie jeden. Jest to spowodowane pracą funkcji, przy przechodzeniu do folderów ręcznie musimy również je uwzględniać.

**B os.chdir()** – ta funkcja służy do zmiany katalogu, w którym się znajdujemy. Oczywi-

```
>>> import os
>>> os.getcwd()
'C:\\Users\\krzys\\AppData\\Loca
B >>> os.chdir('C:\\Python')
>>> os.getcwd()
'C:\\Python'
>>>
```

ście, aby przejść do konkretnej lokalizacji, musi ona istnieć – nie możemy przejść do folderu, który nie istnieje.

```
>>> os.getcwd()
'C:\\Python'
>>> os.makedirs('Folder1') C
>>> os.makedirs('C:\\Python\\Folder1\\Folder2')
>>>
```

```
| Folder1
zędzia główne   Udostępnianie   Widok
Ten komputer > Dysk lokalny (C:) > Python > Folder1 >
Nazwa                      Data modyfikacji
Folder2                    23.06.2021 14:15
```

**C os.makedirs()** – ta funkcja umożliwia tworzenie nowych folderów. Domyślnie foldery tworzone są w lokalizacji, w której się aktualnie znajdujemy. Możemy jednak dokładnie określić ścieżkę dla nowego folderu, przekazując ją bezpośrednio w argumencie funkcji.

```
>>> os.path.abspath('Folder1') D
'C:\\Python\\Folder1'
>>> |
```

**D os.path.abspath()** – dzięki tej funkcji możemy uzyskać dokładną ścieżkę dostępu do podanego w funkcji argumentu.

**E os.path.basename()** – ta funkcja umożliwia uzyskanie nazwy pliku końcowego, który został przypisany do zmiennej zawierającej plik. Przypisywanie zmiennej

```
'C:\\Python\\Folder1'
>>> file_path = "C:\\Python\\Folder1\\plik.txt"
>>> os.path.basename(file_path) E
'plik.txt'
>>>
```

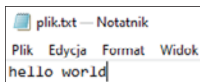
```
Udostępnianie   Widok
Ten komputer > Dysk lokalny (C:) > Python > Folder1
Nazwa                      Data modyfikacji
Folder2                    23.06.2021 14:15
plik.txt                   23.06.2021 14:23
```

```
plik.txt
>>> os.path.split(file_path) F
('C:\\Python\\Folder1', 'plik.txt')
>>> folder,plik = os.path.split(file_path)
>>> print("Plik znajduje się w lokalizacji: ", folder, " i nazywa się: ", plik)
Plik znajduje się w lokalizacji:  C:\Python\Folder1  i nazywa się:  plik.txt
>>> |
```

wartości pliku jest bardzo proste – wystarczy wskazać ścieżkę do pliku na naszym dysku. Pamiętajmy, aby wcześniej utworzyć fizycznie pusty plik tekstowy w wybranej lokalizacji i odpowiednio go nazwać.

**F** **os.path.split()** – pozwala uzyskać ścieżkę dostępu do pliku i nazwę pliku w dwóch osobnych danych. Możemy je przypisać do dwóch różnych zmiennych w celu dalszego przetwarzania.

**G** **os.path.getsize()** – ta funkcja umożliwia pobranie rozmiaru pliku w bajtach. Do naszego




przykładowego pliku wprowadziliśmy tekst hello world, zajmuje on 11 znaków razem ze spacją, która również jest znakiem i tyle właśnie bajtów ma przykładowy plik.

```
>>> print("Plik znajduje się w
Plik znajduje się w lokalizacji
>>> os.path.getsize(file_path) G
11
>>>
```

```
11
>>> os.path.exists("plik.txt")
False
>>> os.getcwd()
'C:\\Python'
>>> os.path.exists("C:\\Python\\Folder1\\plik.txt") H
True
>>>
```

**H** **os.system.exists()** – dzięki tej funkcji otrzymamy wartość logiczną **True** lub **False** w zależności od tego, czy plik lub folder istnieje.

Teraz, gdy znamy już podstawowe funkcje z modułu **os**, możemy przystąpić do pracy z plikami w Pythonie.

## Praca z plikami

W przypadku pracy z plikami nadal będziemy korzystać z interaktywnej sesji w IDLE, dopiero w późniejszym etapie, gdy będziemy tworzyć złożone skrypty, skorzystamy z IDE PyCharm.

## Otwieramy, czytamy i zamykamy plik

**1** W celu otworzenia pliku wpisujemy polecenie **file = open('plik.txt')**. Należy jednak pamiętać: aby móc otworzyć plik, musi on istnieć i musimy podać do niego dokładną ścieżkę dostępu, jeśli nie znajduje się w folderze, w którym aktualnie pracujemy.

```
True
>>> file = open('plik.txt')
Traceback (most recent call last):
  File "<pyshell#17>", line 1, in <module>
    file = open('plik.txt')
FileNotFoundError: [Errno 2] No such file or directory: 'plik.txt'
>>> |
```

**2** Podajemy więc instrukcję poprawnie, pamiętając o ścieżce do pliku. W przypadku, gdy nie podamy drugiego argumentu do funkcji **open**, który określa tryb odtwarzania pliku, zostanie on odtworzony jako tylko do odczytu.

```
    file = open('plik.txt')
FileNotFoundError: [Errno 2] No such file or dire
>>> file = open('C:\\Python\\Folder1\\plik.txt')
>>> |
```

**3** W celu odczytania całego pliku i wyświetlenia go możemy skorzystać z instrukcji **file.read()**.

```
>>> file = open('C:\\
>>> file.read()
'hello world'
>>>
```

**4** Zawsze pamiętajmy, aby po zakończeniu korzystania z plików je zamykać. Służy do tego polecenie **file.close()**.

```
'hello world'
>>> file.close()
>>> file.read()
Traceback (most recent call last):
  File "<pyshell#21>", line 1, in <module>
    file.read()
ValueError: I/O operation on closed file.
>>> |
```

**5** Należy również pamiętać, że w naszym przykładzie **file** to zmienna typu **plik**, która ma przypisaną nazwę. Możemy tworzyć różne zmienne typu **plik** z różnymi nazwami, nie jest to jedyna dostępna i akceptowalna nazwa.

## Korzystamy z trybu zapisu

**1** W celu zapisania jakichkolwiek danych do pliku musimy otworzyć go w trybie do zapisu. Podajemy więc instrukcję **zapis = open('plik.txt','w')**

```
>>> os.chdir("C:\\Python\\Folder1"
        )
>>> os.getcwd()
'C:\\Python\\Folder1'
>>> zapis = open('plik.txt','w')
>>>
```

**2** Warto wiedzieć, że gdy otworzymy plik w trybie zapisu, nie mamy możliwości odczytania jego zawartości. Przy wykonaniu takiej próby pojawi się błąd z odpowiednią informacją.

```
>>> zapis = open('plik.txt','w')
>>> zapis.read()
Traceback (most recent call last):
  File "<pyshell#26>", line 1, in <module>
    zapis.read()
io.UnsupportedOperation: not readable
>>>
```

**3** Po otworzeniu pliku w trybie do zapisu tworzymy dodatkową zmienną – będzie ona przechowywała tekst, który zamierzamy dodać do naszego pliku. Następnie, korzystając z funkcji **write**, dodajemy tekst do naszego pliku.

```
io.UnsupportedOperation: not readable
>>> tekst = "Dodajemy tekst do pliku"
>>> zapis.write(tekst)
23
>>>
```

plik.txt — Notatnik

Plik Edycja Format Widok Pomoc

**4** W chwili tej operacji z poziomu Windows nie widzimy zmian, jakie zachodzą w pliku. Dopiero po zamknięciu pliku – za pomocą polecenia **zapis.close()** – będziemy mogli zauważyć zmiany.

```
io.UnsupportedOperation: not readable
>>> tekst = "Dodajemy tekst do pliku"
>>> zapis.write(tekst)
23
>>> zapis.close()
>>>
```

plik.txt — Notatnik

Plik Edycja Format Widok Pomoc

Dodajemy tekst do pliku

**5** Tekst został prawidłowo dodany do pliku. Niestety, dodawanie danych do pliku z wykorzystaniem tej metody całkowicie usunęło poprzednio zapisane w nim dane. Jak pamiętamy, w tym pliku była zapisana treść: hello world. Została ona bezpowrotnie utracona.

Jak widać, tryby pracy z plikami w Pythonie są kluczowe – wystarczy otworzyć plik w nieodpowiednim trybie i dodać do niego kawałek tekstu, a w rezultacie tracimy wszystkie dane. Na szczęście istnieje jeszcze jeden tryb otwierania pliku.

## Korzystamy z trybu dodawania danych

**1** W celu dodania danych do istniejącego już pliku musimy otworzyć go w trybie **"a"** – **append**, czyli dodawania.

```
NameError: name 'dodaj' is not defined
>>> dodaj = open('plik.txt','a')
>>>
```

**2** Następnie dodajemy tekst do naszego pliku – nie musimy do tego celu wykorzystywać zmiennej, możemy bezpośrednio przekazać tekst jako argument funkcji. Na koniec pamiętajmy o zamknięciu pliku – wtedy zmiany zostaną zapisane.

```
>>> dodaj.write(' Dodajemy kolejny tekst')
23
>>> dodaj.close()
>>>
```

plik.txt — Notatnik

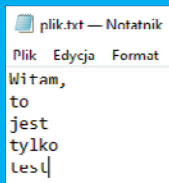
Plik Edycja Format Widok Pomoc

Dodajemy tekst do pliku Dodajemy kolejny tekst

Dodawanie tekstu w tym trybie polega na doklejaniu znaków od miejsca ostatniego zapisanego znaku istniejącego już w pliku.

# CZYTANIE PLIKU TEKSTOWEGO WIERSZ PO WIERSZU

**1** Modyfikujemy ręcznie nasz plik tekstowy, tak aby znajdowały się w nim przynajmniej cztery wiersze tekstu.




**2** Następnie zapisujemy zmiany w pliku i otwieramy plik w trybie do odczytu, korzystając z kodu Pythona.

```
>>> plik = 'plik.txt'
>>> wiersze = open(plik,'r')
>>> |
```

**3** Teraz w celu wyświetlenia zawartości pliku możemy skorzystać ze znanej już funkcji **read**.

```
>>> wiersze.read()
'Witam,\nto\njest\ntylko\ntest'
>>> |
```

**4** Jak widać, domyślnie wszystkie znaki wyświetlane są bez żadnych przerw, a utworzone przez nas przejścia do kolejnych wierszy zostały oznaczone w standardowy sposób dla znacznika nowej linii **\n**. Jeśli chcemy wyświetlić dane bez tego znacznika, musimy skorzystać z pętli.

```
>>> wiersze = open(plik,'r')
>>> for i in wiersze.readlines():
        print(i)


Witam,

to

jest

tylko

test
>>>
```

**5** W powyższym przykładzie widać, że funkcja **readlines** pozwala na odczytywanie kolejnych wierszy. Niestety, znak nowej linii również jest interpretowany. W celu wyeliminowania tego problemu musimy odpowiednio sformatować zwracane dane.

```
>>> wiersze.close()
>>> wiersze = open(plik,'r')
>>> for i in wiersze.readlines():
        i = i.strip('\n')
        print(i)

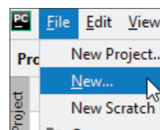

Witam,
to
jest
tylko
test
>>> |
```

**6** Dzięki wykorzystaniu metody **strip** możemy pozbyć się dowolnego znaku z naszego łańcucha. Eliminując znak nowej linii, możemy uzyskać czytelny dla użytkownika tekst podzielony na wiersze.

## Tworzymy plik za pomocą skryptu i zapisujemy do niego dane

Do tej pory na potrzeby przykładów sami tworzyliśmy plik tekstowy w odpowiedniej lokalizacji, aby móc wykonywać różne operacje na konkretnym pliku. Teraz stworzymy skrypt w Pythonie, który będzie przyjmował od użytkownika lokalizację dla pliku, następnie jego nazwę, po czym utworzy plik i zapisze do niego przekazane przez użytkownika dane. Do tego zadania skorzystamy z IDE PyCharm.

**1** Zaczynamy od utworzenia nowego pliku w PyCharm, następnie importujemy moduł **os**, który pozwoli nam na zmianę folderu pracy na ten, w którym będzie znajdował się docelowy plik. Klikamy na górnym pasku na **File**, **New**, wybieramy plik Pythona, nadajemy mu nazwę i importujemy moduł **os**.




**2** Następnie tworzymy zmienną, która przyjmie od użytkownika lokalizację i zmieni nasz aktualny folder na tę lokalizację, po czym pobieramy od użytkownika informację, jaka jest nazwa pliku.

```
folder = input('Podaj ścieżkę do folderu '
               '(Użyj \\\ do oddzielenie folderów):')
folder = folder.strip('\n')
os.chdir(folder)
plik = input("Podaj nazwę pliku do utworzenia: ")
```

**3** Na koniec podajemy treść, jaka ma być dodawana do pliku, lub pobieramy ją

od użytkownika. Potem, korzystając z trybu **'wt'**, otwieramy plik do zapisu tekstu, co automatycznie tworzy nowy plik o wskazanej nazwie. Po zapisie – używamy funkcji **write** – zamykamy plik, aby zmiany zostały zachowane.

```
zawartosc = "Witaj!\nTo tylko\ntestowa\ninformacja"
x = open(plik, 'wt')
x .write(zawartosc)
x.close()
```

**4** Należy pamiętać o tym, aby w odpowiedni sposób podawać ścieżkę do katalogu, a podając nazwę pliku, dodawać rozszerzenie, jakie powinien mieć plik. W naszym przykładzie będzie to **txt**, ponieważ zamierzamy do naszego pliku dodać tekst. Po zakończeniu działania skryptu zostanie utworzony nowy plik z odpowiednią treścią.

## Odwracanie tekstu odczytywanego z pliku

W poprzednim rozdziale napisaliśmy skrypt, który umożliwia odwracanie ciągu znaków. Teraz utworzymy skrypt, który pozwoli na wykonywanie podobnego zadania podczas pracy z plikami. Zadaniem będzie odczytanie pliku wiersz po wierszu, odwrócenie tekstu i zapisanie odwróconego tekstu w miejsce początkowego tekstu. Przy zapisywaniu pliku cały tekst musi zostać odwrócony.

**1** Zaczynamy od wykorzystania kodu z poprzedniego skryptu. Tym razem jednak – dla ułatwienia i przyspieszenia całego procesu – wewnątrz skryptu definiujemy lokalizację folderu oraz pliku, na którym zamierzamy pracować. Otwieramy go w trybie do odczytu.

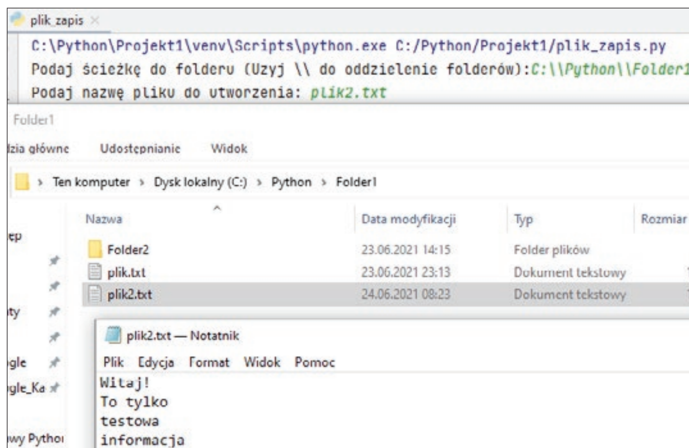
```
import os

os.chdir('C:\\Python\\Folder1')
plik = 'plik2.txt'
x = open(plik, 'r')
```

**2** Teraz ponownie tworzymy pętlę **for**, która umożliwi nam odczytanie wiersz po wierszu pliku tekstowego. Pamiętajmy o funkcji **strip**, aby pozbyć się nadmiaru nowych linii, oraz szybkim odwracaniu ciągu znaków z wykorzystaniem cechy **wycinka**. Po pobraniu danych w trybie odczytu musimy zamknąć plik, aby otworzyć go ponownie w trybie do zapisu. W tym trybie dokonujemy zapisu odwróconego ciągu znaków do konkretnego wiersza. Na koniec musimy zamknąć plik, aby zmiany zostały zachowane, i ponownie otworzyć go w trybie do odczytu, aby przy kolejnej iteracji pętli móc odczytać kolejny wiersz.

```
for i in x.readlines():
    i = i.strip('\n')
    i = i[::-1]
    print(i)
    x.close()
    x = open(plik,'w')
    x.write(i)
    x.close()
    x = open(plik, 'r')

x.close()
```



**3** Jednak ponownie przez wykorzystanie trybu **zapisu**, za każdym razem, gdy do pliku dodawany jest tekst, usuwane są pozostałe wiersze. Mimo że w konsoli skrypt podał nam wszystkie odwrócone ciągi znaków, w pliku znalazł się tylko ostatni ciąg.

```
plik_zapis
C:\Python\Projekt1\venv\Scripts\py
!jatiW
oklyt oT
awotset
ajcamrofni

Process finished with exit code 0
```

```
plik2.txt — Notatnik
Plik  Edycja  Format  Widok  Pomoc
ajcamrofni
```

**4** Aby to naprawić, musimy skorzystać z trybu **dodawania**. Jeżeli chcemy otrzymać tekst w tym samym pliku dopisany do jego końca, wystarczy, że zmienimy tryb **'w'** na tryb **'a'**. Jeśli chcemy otrzymać nowy plik z odwróconym tekstem, dodajemy nową zmienną z innym plikiem i podajemy go w kodzie.

```
import os

os.chdir('C:\\Python\\Folder1')

plik_a = 'plik2.txt'
plik_b = 'plik.txt'
x = open(plik_a, 'r')

for i in x.readlines():
    i = i.strip('\n')
    i = i[::-1]
    print(i)
    x.close()
    x = open(plik_b, 'a')
    x.write(i)
    x.close()
    x = open(plik_a, 'r')

x.close()
```

**5** Po uruchomieniu skryptu cały tekst został odwrócony wiersz po wierszu, jednak został on zapisany w jednej linii. W celu oddzielenia kolejnych wierszy, musimy ręcznie dodać znak nowej linii.

```
skrypt2
C:\Python\Projekt1\venv\Scripts\pytho
!jatiW
oklyt oT
awotset
ajcamrofni

Process finished with exit code 0
```

```
plik.txt — Notatnik
Plik  Edycja  Format  Widok  Pomoc
!jatiWoklyt oTawotsetajcamrofni
```

**6** Po instrukcji otwierania pliku w trybie dodawania wstawiamy kod, który służy do dodawania znaku nowej linii na końcu łańcucha znaków.

```
    print(i)
    x.close()
    x = open(plik_b, 'a')
    i = i + '\n'
    x.write(i)
    x.close()
    x = open(plik_a, 'r')
```

**7** Teraz po uruchomieniu skryptu uzyskamy odpowiednio sformatowany odwrócony tekst odczytany z jednego pliku i zapisany w drugim pliku.

```
Run:  skrypt2
C:\Python\Projekt1\venv\Scripts\pyt
!jatiW
oklyt oT
awotset
ajcamrofni
```

```
plik.txt — Notatnik
Plik  Edycja  Format  Widok
!jatiW
oklyt oT
awotset
ajcamrofni
```

```
plik2.txt — Notatnik
Plik  Edycja  Format  Widok
Witaj!
To tylko
testowa
informacja
```

# Pracujemy z datą oraz czasem w Pythonie

Bardzo często w różnego rodzaju skryptach przydaje się możliwość odwoływania się do dat i czasu. Aktualna data i czas mogą być nam niezbędne, na przykład gdy tworzymy skrypt, który będzie wysyłał regularnie informacje o stanie komputera w określonych odstępach czasu. Praktycznie wszystkie niezbędne funkcje związane z datą i czasem znajdziemy we wbudowanym module **datetime** – czas poznać go nieco lepiej. By poznać różne funkcje, skorzystamy z interaktywnej sesji **IDLE**.

## Uzyskujemy aktualne datę i godzinę

1 Po zaimportowaniu modułu **datetime** tworzymy obiekt, który będzie przechowywał dane na temat daty, i przypisujemy do niego funkcję **datetime.datetime.now()**.

```
'datetime_CAPI', 'sys', 'time', 'time
>>> data_1 = datetime.datetime.now()
>>> print(data_1)
2021-06-24 13:13:27.159953
>>>
```

## KLASY I PROGRAMOWANIE OBIEKTOWE

W dalszej części książki zapoznamy się z klasami i obiektami w Pythonie. Można powiedzieć, że klasy służą dostosowywaniu obiektów do naszych potrzeb dzięki dziedziczeniu. Są one w gruncie rzeczy pakietami funkcji, które pozwalają na przetwarzanie i wykorzystywanie wbudowanych typów. Klasami w przykładach na najbliższych stronach będzie data oraz czas. Z pewnością będziemy mogli łatwo zauważyć różnicę w sposobie odwoływania się do obiektów w stosunku do tego, jak pracowaliśmy do tej pory ze zwykłymi zmiennymi.
Jeśli jakiś fragment jest mało zrozumiały na tym etapie – nie przejmujmy się! W dalszej części książki koncepcja klasy i programowania obiektowego zostanie bardziej rozwinięta.

## WARTO WIEDZIEĆ!

**print(dir(datetime)** – ta instrukcja pozwala uzyskać informacje o funkcjach dostępnych wewnątrz modułu. Należy zwrócić uwagę, że moduł **datetime** zawiera sam w sobie funkcję **datetime**, co na początku może być nieco mylące. Aby móc skorzystać z tego polecenia, musimy wcześniej wykonać import tego modułu.

IDLE Shell 3.9.5

File Edit Shell Debug Options Window Help

```
Python 3.9.5 (tags/v3.9.5:0a7dcbd, May  3 2021, 17:27:52) [MSC v.1928 64 bit (AM
D64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> import datetime
>>> print(dir(datetime))
['MAXYEAR', 'MINYEAR', '__all__', '__builtins__', '__cached__', '__doc__', '__fi
le__', '__loader__', '__name__', '__package__', '__spec__', 'date', 'datetime',
'datetime_CAPI', 'sys', 'time', 'timedelta', 'timezone', 'tzinfo']
>>> |
```

**2** Warto zapamiętać, że aktualna data i godzina zapisywane są do obiektu w momencie wywołania funkcji i z takim znacznikiem czasowym pozostają do czasu ręcznej zmiany ich wartości.

```
>>> data_2 = datetime.datetime.now()
>>> print(data_2)
2021-06-24 13:14:20.056456
>>> print(data_1)
2021-06-24 13:13:27.159953
>>>
```

**3** Jeśli chcemy uzyskać tylko i wyłącznie informację o aktualnej dacie, należy skorzystać z funkcji **datetime.date.today()**.

```
>>> data_1 = datetime.date.today()
>>> print(data_1)
2021-06-24
>>> |
```

W celu utworzenia obiektu **data** możemy również sami przekazać odpowiednie parametry do funkcji **date**. Wystarczy przekazać trzy całkowite liczby, które mają reprezentować datę, czyli kolejno: rok, miesiąc, dzień.

```
2021-06-24
>>> a = 2021
>>> b = 6
>>> c = 25
>>> data_3 = datetime.date(a, b, c)
>>> print(data_3)
2021-06-25
>>> |
```

## Korzystamy ze znaczników czasu

Znaczniki czasu w Pythonie pozwalają na dokładne wskazanie daty po liczbie sekund liczonej od 1 stycznia 1970 roku. Bardzo często jest to przydatna opcja, gdyż, żeby zaprezentować datę, możemy operować na tylko jednej zmiennej.

```
>>> znacznik_1 = date.fromtimestamp(1)
>>> print('Data ze znacznika: ', znacznik_1)
Data ze znacznika:  1970-01-01
>>> znacznik_1 = date.fromtimestamp(8230213103)
>>> print('Data ze znacznika: ', znacznik_1)
Data ze znacznika:  2230-10-22
>>> |
```

## POMOCNE IMPORTOWANIE

**Warto pamiętać o możliwości importu konkretnej funkcji lub klasy z danego modułu, dzięki temu nie będziemy musieli bez przerwy się do niego odwoływać. Dla przykładu, ponieważ często korzystamy z klasy date, importujemy ją bezpośrednio.**

```
>>> from datetime import date
>>> dzisiaj = date.today()
>>> print(dzisiaj)
2021-06-24
>>> |
```

## Wyświetlamy konkretne dane dotyczące daty

Korzystając z funkcji **today** **A**, możemy uzyskać dane dotyczące aktualnej daty – może to być rok, miesiąc lub dzień. Wystarczy przypisać aktualną datę do konkretnej zmiennej, a następnie odwoływać się do konkretnych atrybutów.

## Zapisujemy czas jako obiekt

W powyższych przykładach zajmowaliśmy się zapisywaniem daty jako obiektu. Teraz skupimy się na czasie.

**1** Importujemy z modułu **timestamp** klasę **time**.

```
IDLE Shell 3.9.5
File  Edit  Shell  Debug  Options  Window
Python 3.9.5 (tags/v3.9.5:0a7dcbd
D64)] on win32
Type "help", "copyright", "credit:
>>> from datetime import time
>>>
```

**2** Następnie tworzymy obiekt typu **time**. Domyślnie będzie on pusty i będzie zawierał czas określony w domyślnym kon-

```
>>> dzisiaj = date.today()   A
>>> print(dzisiaj)
2021-06-24
>>> print("Rok: ", dzisiaj.year, "Miesiąc: ", dzisiaj.month, "Dzień: ", dzisiaj.day)
Rok:  2021 Miesiąc:  6 Dzień:  24
>>> |
```

struktorze klasy, w tym przypadku jest to **00:00:00**.

```
>>> from datetime import time
>>> a = time()
>>> print("a = ", a)
a =  00:00:00
>>>
```

**3** Tworząc obiekt klasy **time**, możemy podać dla niego parametry wejściowe. W kolejności – godziny, minuty, sekundy.

```
>>> b = time(12,34,56)
>>> print("b = ", b)
b =  12:34:56
>>>
```

**4** Podobnie jak w przypadku daty, do obiektów utworzonych dla czasu możemy odwoływać się, wywołując każdy z ich atrybutów osobno. Wystarczy odwołać się w kodzie do konkretnego atrybutu, wskazując uprzednio odpowiedni obiekt.

```
>>> c = time(12,13,14,2412)
>>> print(c)
12:13:14.002412
>>> print(c.hour)
12
>>> print(c.minute)
13
>>> print(c.second)
14
>>> print(c.microsecond)
2412
>>> |
```

## Obliczamy różnicę pomiędzy dwiema datami

Do tego zadania będziemy potrzebować klasy **datetime** oraz **date** z modułu **datetime**. Jest to jedna z bardziej przydatnych operacji. Dzięki obliczeniu różnicy pomiędzy dwiema datami możemy na przykład utworzyć kalkulator dni do końca roku, do naszych urodzin lub po prostu policzyć różnicę wyrażoną w dniach pomiędzy dwiema datami.

**1** Zaczynamy od importu odpowiednich klas z modułu **datetime**. Warto pamiętać, że możemy po przecinku podawać kolejne funkcje lub klasy do zaimportowania z konkretnego modułu.

```
IDLE Shell 3.9.5
File  Edit  Shell  Debug  Options  Window  Help
Python 3.9.5 (tags/v3.9.5:0a7dcbd, May  3 2021
D64)] on win32
Type "help", "copyright", "credits" or "licens
>>> from datetime import datetime, date
>>>
```

**2** Następnie tworzymy dwa obiekty typu **date**, którym przypisujemy konkretne wartości.

```
>>> from datetime import datetime, dat
>>> t1 = date(2021,6,24)
>>> print(t1)
2021-06-24
>>> t2 = date(2021,12,31)
>>> print(t2)
2021-12-31
>>> |
```

**3** Teraz tworzymy trzeci obiekt pomocniczy, który przechowa różnicę pomiędzy utworzonymi wcześniej obiektami. Ponieważ obiekty są tego samego typu i zdefiniowana jest dla nich obsługa operatora różnicy, możemy odjąć je od siebie.

```
>>> print(t2)
2021-12-31
>>> t3 = t2 - t1
>>> print(t3)
190 days, 0:00:00
>>>
```

**4** Możemy również, korzystając z atrybutów dla obiektu **date**, otrzymać samą wartość liczbową dni uzyskaną w wyniku odejmowania dwóch dat.

```
>>> print(t3)
190 days, 0:00:00
>>> print(t3.days)
190
```

Jeśli potrzebujemy dokładnej różnicy pomiędzy dwiema datami z dokładnością do sekund, najlepiej skorzystać z klasy **timedelta**. Importujemy ją **A** i po utworzeniu niezbędnych obiektów **B** możemy odejmować od siebie różnego typu daty. Nie muszą one być

```
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <modu
    print(t3.date())
AttributeError: 'datetime.timedelta' o
>>> from datetime import timedelta  A
```

```
>>> t1 = timedelta(weeks = 3, days = 5, hours = 3, seconds = 23)  B
>>> print(t1)
26 days, 3:00:23
>>> t2 = timedelta(days = 9, hours = 8, minutes = 44, seconds = 23)
>>> print(t2)
9 days, 8:44:23
>>>
```

```
>>> t2 = timedelta(da
>>> print(t2)
9 days, 8:44:23
>>> r3 = t1 - t2
>>> print(r3)
16 days, 18:16:00
>>>
```

```
>>> print(t4)
-17 days, 5:44:00
>>> |
```

tak samo sformatowane, a obliczenia i tak zostaną wykonane. Jeśli chcemy, możemy uzyskać negatywny wynik, który będzie informował o tym, ile dni upłynęło od konkretnej daty.

## Różne formaty daty

Standardowy format daty zwracany przez funkcję **now** z klasy **datetime** nie zawsze jest najbardziej odpowiedni do naszych potrzeb. Możemy chcieć uzyskać dni, miesiąc i rok w innej kolejności lub oddzielone innym separatorem, na przykład kropką, tak jak robi to Windows 10, lub /, jak często jest w oficjalnych dokumentach.

**1** Zaczynamy standardowo od zaimportowania odpowiedniej klasy i modułu – w tym wypadku będzie to **datetime** z modułu **datetime**. Domyślnie data sformatowana jest tak jak w przypadku wywołania metody **now()**.

```
IDLE Shell 3.9.5
File  Edit  Shell  Debug  Options  Window  H
Python 3.9.5 (tags/v3.9.5:0a7dcbd,
D64)] on win32
Type "help", "copyright", "credits'
>>> from datetime import datetime
>>> teraz = datetime.now()
>>> print(teraz)
2021-06-25 09:57:26.774693
>>> |
```

**2** Możemy jednak, korzystając z funkcji **strftime**, przekazać potrzebny nam format jako argument funkcji.

```
>>> t = teraz.strftime('%H:%M:%S')
>>> print(t)
09:57:26
>>> |
```

**3** Należy jednak stosować się do dopuszczalnej przez funkcję składni. Ma ona bardzo dużo różnego typu formatów, które wspiera – w celu zapoznania się z nimi warto sprawdzić informacje na jej temat w dokumentacji Pythona.

```
>>> t2 = teraz.strftime('%d.%m.%Y, %H:%M:%S')
>>> print(t2)
25.06.2021, 09:57:26
>>> |
```

## Wyrażenia regularne

Python obsługuje wyrażenia regularne, są to pewne wzorce, za pomocą których można opisać ciąg znaków. Dzięki nim możemy uzyskać ważne dane w bardzo prosty sposób. Wyobraźmy sobie, że mamy do dyspozycji bardzo obszerne dane tekstowe zawierające mnóstwo różnego typu informacji, w tym na przykład numer telefonu lub adres e-mail. Korzystając właśnie z wyrażeń regularnych, czyli **REGEXP**, możemy szybko pozyskać potrzebne nam dane.

**1** Wszystkie klasy i funkcje związane z wyrażeniami regularnymi znajdziemy we wbudowanym module **re**. Musimy go więc zaimportować do naszego kodu. Ponieważ przykłady będą nieco bardziej złożone, będziemy bazować na IDE PyCharm, by się z nimi zapoznać.

## WARTO WIEDZIEĆ!

Korzystając z funkcji **strptime**, możemy również przekonwertować datę zapisaną w formacie ciągu znaków na obiekt daty, który możemy przetwarzać dalej i porównywać z innymi datami.

```
>>> data_string = '22 May 2021'
>>> print(data_string)
22 May 2021
>>> data_obiekt = datetime.strptime(data_string, '%d %B %Y')
>>> print(data_obiekt)
2021-05-22 00:00:00
>>> |
```

**2** Załóżmy, że chcemy uzyskać numer telefonu z podanego łańcucha znaków.

```
import re

tekst = """Witam, to jest bardzo długi ciąg znaków
z ukrytym w nim numerem telefonu, teraz musimy znaleźć
ten numer 666-777-888 korzystając z wyrażeń regularnych"""
```

**3** W tym celu tworzymy wzór dopasowania do szukanego wyrażenia. Początek z literą **r** oznacza, że jest to wyrażenie regularne, a wewnątrz jest wzorzec. Oznaczenie **\d** informuje, że poszukujemy dowolnej jednej cyfry od 0 do 9.

```
z ukrytym w nim numerem telefonu, t
ten numer 666-777-888 korzystając z

wzor = r'\d\d\d-\d\d\d-\d\d\d'
```

**4** Następnie, korzystając z funkcji **search**, wskazujemy jako atrybuty ciąg znaków do przeszukania oraz wzorzec, którego poszukujemy. W kolejnym kroku korzystamy z funkcji **group()**, która pozwala na uzyskanie czytelnego wyniku i przypisujemy znalezioną wartość do zmiennej, którą na koniec wyświetlamy.

```
import re

tekst = """Witam, to jest bardzo długi ciąg znaków
z ukrytym w nim numerem telefonu, teraz musimy znaleźć
ten numer 666-777-888 korzystając z wyrażeń regularnych"""

wzor = r'\d\d\d-\d\d\d-\d\d\d'

szukaj = re.search(wzor, tekst)

numer = szukaj.group()

print ("Znaleziony numer to: ", numer)
```

```
C:\Python\Projekt1\venv\Scripts\python.exe C:/Python/Projekt1/skrypt2.py
Znaleziony numer to:  666-777-888

Process finished with exit code 0
```

Podsumowując: w linijce **wzor = r'\d\d\d-\d\d\d-\d\d\d'** utworzyliśmy wyrażenie regularne. Znak **r** przed **'** oznacza, że zwykłe formatujące działanie \ jest wyłączone. Jest to niezwykle istotne, gdyż bez tego nie moglibyśmy zbudować wyrażenia regularnego.

Następnie, korzystając z funkcji **search**, wyszukujemy pasujący do wzorca ciąg znaków. Funkcja ta zwraca wynik jako obiekt typu **Match**. Nie moglibyśmy go normalnie używać, dlatego też konieczne jest skorzystanie z funkcji **group**, która konwertuje ten obiekt do zwykłego łańcucha znaków.

W tabelce poniżej znajdziemy informacje na temat wyrażeń regularnych, które pozwolą na tworzenie skomplikowanych wzorców.

## SPECJALNE KLASY ZNAKÓW

| ZNAK | OPIS |
|---|---|
| \s | Spacja, tabulator lub znak nowego wiersza |
| \S | Znak, który nie jest spacją, tabulatorem lub znakiem nowego wiersza |
| \w | Litera, cyfra lub znak "_" (zapis równoważny [A-Za-z_]) |
| \W | Znak, który nie jest literą, cyfrą lub "_" |
| \d | Cyfra (zapis równoważny [0-9]) |
| \D | Znak, który nie jest cyfrą |
| \b | Dowolny znak odstępu, dopasowywanie początku lub końca słowa |

Szczegóły dotyczące wyrażeń regularnych można sprawdzić również w dokumentacji dotyczącej Pythona.

## Uzyskujemy wszystkie znalezione dane pasujące do wzorca

W poprzednim przykładzie korzystaliśmy z funkcji **search**. Działa ona bardzo dobrze, jednak służy do znajdowania tylko pierwszego elementu pasującego do wzorca w całym ciągu. Znaczy to, że jeśli w danym ciągu jest kilka pasujących elementów, my uzyskamy tylko jeden. Jeżeli zależy nam na liście wszystkich elementów, musimy skorzystać z funkcji **findall**. Na potrzeby tego przykładu należy utworzyć plik tekstowy zawierający adresy e-mail w kolejnych wierszach i inne teksty.

```
mail.txt — Notatnik
Plik  Edycja  Format  Widok  Pomoc
witam
asddad@dsada.com
dsadad@dfs.com
fdafgdf sada a
sad


cccc@ccccc.com
dddd@ddddc.com
```

**1** Po utworzeniu pliku i wpisaniu do niego danych rozpoczynamy od importu modułów **re** oraz **os**. Przechodzimy do folderu, w którym znajduje się plik, i przypisujemy go do obiektu.

```
main.py × | skrypt2.py × | plik_zapis.py ×
import re, os

os.chdir("C:\\Python\\Folder1")
plik = "mail.txt"
```

**2** Następnie otwieramy plik w trybie do odczytu, umieszczamy cały tekst w buforze, a następnie, korzystając z funkcji **findall** **A**, podajemy jako jeden z argumentów wzorzec do znalezienia adresu e-mail oraz tekst do przeszukania. Na koniec wyświetlamy wyniki.

```
x = open(plik, 'r')
tekst = x.read()
                              A
dopasuj = re.findall(r'[\w.-]+@[\w.-]+',tekst)
print(dopasuj)
```

```
skrypt2 ×
C:\Python\Projekt1\venv\Scripts\python.exe C:/Python/Projekt1/skrypt2.py
['asddad@dsada.com', 'dsadad@dfs.com', 'cccc@ccccc.com', 'dddd@ddddc.com']

Process finished with exit code 0
```

**3** Warto zauważyć, że w tym przypadku nie musieliśmy korzystać z funkcji **group**, aby wyniki były odpowiednio prezentowane.

## Rozbijamy ciąg znaków na pojedyncze znaki

Czasem możemy potrzebować uzyskać listę znaków, które tworzą ciąg. Jak wiemy, ciągi znaków są niezmienne – listy natomiast jak najbardziej możemy modyfikować.
Dzięki wyrażeniom regularnym to zadanie staje się dość proste.

```
IDLE Shell 3.9.5
File  Edit  Shell  Debug  Options  Window  Help
Python 3.9.5 (tags/v3.9.5:0a7dcbd, May  3 2021, 17:27:52) [MSC v.1928 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> import re  B
>>> string = "Dzisiaj jest czwartek"
>>> regex = re.compile(r'[a-zA-Z0-9]')
>>> regex.findall(string)
['D', 'z', 'i', 's', 'i', 'a', 'j', 'j', 'e', 's', 't',
'c', 'z', 'w', 'a', 'r', 't', 'e', 'k']
>>>
```

Po zaimportowaniu odpowiedniego modułu **B** tworzymy ciąg znaków, a następnie, korzystając z funkcji **compile**, tworzymy wzorzec, który będzie pasował do wszystkich znaków. Następnie korzystamy z funkcji **findall**, która znajdzie każdy z naszych znaków. W tym przykładzie zastosowane zostało wyrażenie regularne, które dodatkowo pomija białe znaki, takie jak spacja.
Jeśli chcemy poćwiczyć z wyrażeniami regularnymi, warto korzystać ze strony o adresie **pythex.org**

# pythex

Your regular expression:

/ [A-Z] /

IGNORECASE MULTILINE DOTALL VERBOSE

Your test string:

Dzisiaj jest czwartek

Match result:

Dzisiaj jest czwartek

Match captures:

No groups.

Regular expression cheatsheet

Inspired by Rubular. For a complete reference, see the official re module documentation.
Made by Gabriel Rodriguez. Powered by Flask and jQuery.

Klikając na **Regular expression cheatsheet**, możemy szybko podejrzeć, z jakich znaków i specjalnych symboli możemy skorzystać, tworząc nasz wzorzec.

Regular expression cheatsheet

**Special characters**

| | |
|---|---|
| \ | escape special characters |
| . | matches any character |
| ^ | matches beginning of string |
| $ | matches end of string |
| [5b-d] | matches any chars '5', 'b', 'c' or 'd' |
| [^a-c6] | matches any char except 'a', 'b', 'c' or '6' |
| R\|S | matches either regex R or regex S |
| () | creates a capture group and indicates precedence |

**Quantifiers**

| | |
|---|---|
| * | 0 or more (append ? for non-greedy) |
| + | 1 or more (append ? for non-greedy) |
| ? | 0 or 1 (append ? for non-greedy) |
| {m} | exactly m m occurrences |
| {m, n} | from m to n. m defaults to 0, n to infinity |
| {m, n}? | from m to n, as few as possible |

**Special sequences**

| | |
|---|---|
| \A | start of string |
| \b | matches empty string at word boundary (between \w and \W) |
| \B | matches empty string not at word boundary |
| \d | digit |
| \D | non-digit |
| \s | whitespace: [ \t\n\r\f\v] |
| \S | non-whitespace |
| \w | alphanumeric: [0-9a-zA-Z_] |
| \W | non-alphanumeric |
| \Z | end of string |
| \g<id> | matches a previously defined group |

**Special sequences**

| | |
|---|---|
| (?iLmsux) | matches empty string, sets re.X flags |
| (?:...) | non-capturing version of regular parentheses |
| (?P...) | matches whatever matched previously named group |
| (?P=) | digit |
| (?#...) | a comment; ignored |
| (?=...) | lookahead assertion: matches without consuming |
| (?!...) | negative lookahead assertion |
| (?<=...) | lookbehind assertion: matches if preceded |
| (?<!...) | negative lookbehind assertion |
| (?(id)yes\|no) | match 'yes' if group 'id' matched, else 'no' |

Based on tartley's python-regex-cheatsheet.

# 5 Obsługa błędów w praktyce

**Do tej pory, tworząc skrypty i analizując przykłady, nie zwracaliśmy uwagi na to, czy nasz program zawsze zadziała zgodnie z naszymi intencjami i czy nie pojawią się jakieś komplikacje. Teraz zajmiemy się obsługą wszelkich błędów, jakie mogą się pojawić w naszym programie**

## Korzystamy z instrukcji try i except

W Pythonie każdy kawałek kodu możemy sprawdzać pod kątem wystąpienia w nim błędów na wiele różnych sposobów. Najczęściej obsługę błędów realizuje się, wykorzystując instrukcje **try** oraz **except**. W dużym uproszczeniu: w pierwszym bloku wstawiamy kod naszego programu, a w drugim określamy, co ma się wydarzyć, gdy pojawi się błąd, dodatkowo wskazując, jaki to może być błąd.

Konieczność reagowania na błędy najczęściej jest powiązana z błędami ludzkimi lub interpretacją poleceń. Załóżmy, że tworzymy kalkulator i prosimy użytkownika o podanie dwóch liczb. Kodując kalkulator, spodziewamy się bezpośrednich wartości liczbowych typu 5, 7, 20. Tymczasem użytkownik wprowadza dane typu: pięć, siedem, dwadzieścia. Nasz program nie zadziała, a użytkownikowi pojawi się błąd, który może mu nic nie powiedzieć, zwłaszcza że użytkownik nie musi w ogóle znać się na programowaniu. To programista jest odpowiedzialny za to, aby przewidzieć możliwe błędy, jakie mogą się pojawić, oraz obsłużyć te błędy z odpowiednią informacją zwrotną w taki sposób, aby użytkownik mógł bez problemów korzystać ze skryptu.

Zobaczmy, jak w praktyce wygląda opisany powyżej proces. Posłużymy się przykładem skryptu do obliczania pola koła – jest nieco mniej rozbudowany.

Na tym zrzucie **A** widać cały kod skryptu oraz jego pracę. Po podaniu poprawnej wartości, jakiej oczekujemy do dalszych obliczeń, pole powierzchni jest liczone, a po podaniu znaku **x** program kończy działanie. Zobaczmy, co stanie się, gdy podamy promień jako słowo. Program od razu zakończył działanie z kodem **1** **B**, co oznacza, że pojawił się błąd. Zawsze wyświetlana jest krótka informacja na temat błędu, która w prostych przypadkach jest w stanie naprowadzić nas na problem, w zaawansowanych skryptach nie zawsze będziemy mogli w ciągu paru sekund zauważyć błąd. W tym przypadku mamy jasną informację, że nie udało się dokonać konwersji typu string (ciąg znaków) na float (liczba zmiennoprzecinkowa).

```
Projekt1 C:\Python\
  venv library root
  funkcje.py
  kalkulator.py
  main.py
  minima.py
  odwracanie_ciagu
  plik_zapis.py
  pliki_odwracanie.
  pole_kola.py
  Projekt1.zip
  skrypt2.py
  szukaj_maila.py
  usuwanie samog
  wzorzec1.py
  zgadnij_liczbe.py
External Libraries
Scratches and Conso
```

**A**

```python
# Skrypt do obliczania pola powierzchni koła
import math

def policz_pole_okr(promien):
    if promien == "x":
        exit()
    else:
        promien2 = float(promien)
        pole = math.pi*promien2*promien2
        print("Pole powierzchni wynosi: ", pole)

print("Gdy chcesz zakończyc prace programu wprowadź x jako
while True:
    promien = input("Wprowadz promień okręgu: ")
    policz_pole_okr(promien)
```

```
Run: skrypt2
C:\Python\Projekt1\venv\Scripts\python.exe C:/Python/Projekt1/skrypt2.py
Gdy chcesz zakończyc prace programu wprowadź x jako promień
Wprowadz promień okręgu: 5
Pole powierzchni wynosi:  78.53981633974483
Wprowadz promień okręgu: x

Process finished with exit code 0
```

```
skrypt2
C:\Python\Projekt1\venv\Scripts\python.exe C:/Python/Projekt1/skrypt2.py
Gdy chcesz zakończyc prace programu wprowadź x jako promień
Wprowadz promień okręgu: trzy
Traceback (most recent call last):
  File "C:\Python\Projekt1\skrypt2.py", line 15, in <module>
    policz_pole_okr(promien)
  File "C:\Python\Projekt1\skrypt2.py", line 8, in policz_pole_okr
    promien2 = float(promien)
ValueError: could not convert string to float: 'trzy'

Process finished with exit code 1
```

**B**

Jak widać z powyższego kodu, oczekujemy od użytkownika wprowadzenia liczby, która musi spełniać pewne wymagania. Dodatkowo w celu opuszczenia programu użytkownik również musi wprowadzić konkretny znak. W przypadku wprowadzenia znaku **X** zamiast **x** również pojawi się błąd **C**.

Należy przewidzieć każdy możliwy scenariusz, jaki może spowodować błąd, na przykład wprowadzenie ujemnej

```
Wprowadz promień okręgu: 3
Pole powierzchni wynosi:  28.274333882308138
Wprowadz promień okręgu: X
Traceback (most recent call last):
  File "C:\Python\Projekt1\skrypt2.py", line 15, in <
    policz_pole_okr(promien)
  File "C:\Python\Projekt1\skrypt2.py", line 8, in po
    promien2 = float(promien)
ValueError: could not convert string to float: 'X'

Process finished with exit code 1
```

**C**

```
14      promien = input("Wprowadz promień okręgu: ")
15      policz_pole_okr(promien)
16
```

```
C:\Python\Projekt1\venv\Scripts\python.exe C:/Python/Projekt1/skrypt2.py
Gdy chcesz zakończyc prace programu wprowadź x jako promień
Wprowadz promień okręgu: 5
Pole powierzchni wynosi:  78.53981633974483
Wprowadz promień okręgu: -5   D
Pole powierzchni wynosi:  78.53981633974483
Wprowadz promień okręgu:
```

wartości dla promienia **D**. Co się wtedy stanie? Jaki będzie rezultat działania naszego skryptu? W tym przypadku okazuje się, że wynik jest identyczny jak dla wartości dodatniej – wynika to ze sposobu obliczania pola powierzchni, gdzie jedną ze zmiennych jest promień podniesiony do kwadratu, a jak wiemy, liczba ujemna podniesiona do kwadratu daje liczbę dodatnią. Jednak z punktu widzenia matematyki nie istnieje coś takiego jak ujemny promień. Pewne wartości po prostu muszą być dodatnie. Jest to bardzo dobry przykład błędu ukrytego w kodzie, który nie powoduje krytycznego błędu działania skryptu, jednak sprawia, że nasz kod jest niskiej jakości. Powinniśmy wyłapać źle wprowadzone dane wejściowe i poprosić użytkownika o poprawne dane.

Do tego właśnie celu wykorzystamy bloki **try** oraz **except**. Zanim jednak przejdziemy do poprawienia skryptu, przyjrzyjmy się bardzo prostemu przykładowi bez żadnej pętli.

**1** W tym przykładzie błąd polega na dzieleniu przez zero.

**2** Jak łatwo zauważyć, przypisanie do zmiennej **b** wartości **0** powoduje w wyniku dzielenia błąd krytyczny. Python nadaje dla tego błędu konkretną nazwę **ZeroDivisionError** **E** (w tłumaczeniu: błąd dzielenia przez zero). To właśnie, korzysta-

```
a = 3
b = 0

c = a/b
```

```
C:\Python\Projekt1\venv\Scripts\python.exe C:/Python/Projekt
Traceback (most recent call last):
  File "C:\Python\Projekt1\main.py", line 4, in <module>
    c = a/b
ZeroDivisionError: division by zero   E

Process finished with exit code 1
```

```
try:   F
    a = 3
    b = 0
    c = a/b
except:   G
    print("Pojawił się błąd")
```

```
C:\Python\Projekt1\venv\Scripts\python.exe C:_
Pojawił się błąd

Process finished with exit code 0
```

jąc z tej nazwy błędu, możemy odnieść się do konkretnego typu błędu podczas obsługi kolejnych przypadków krytycznych zwracanych przez nasz program. Możemy również obsłużyć dowolny błąd zwracany przez program, nie wiedząc, jaką dokładnie ma nazwę. Wystarczy kod, który ma być wykonany, ująć w instrukcji **try** **F**, a to, co ma zostać wykonane w przypadku wykrycia błędu – w instrukcji **except** **G**.

```
try:
    a = 3
    b = 0
    c = a/b
except ZeroDivisionError:   H
    print("Błąd dzielenia przez zero")
except:
    print("Pojawił się błąd")
```

```
C:\Python\Projekt1\venv\Scripts\python.exe C:/Python/Projek
Błąd dzielenia przez zero

Process finished with exit code 0
```

**3** Znając dokładnie nazwę błędu, możemy ją podać po instrukcji **except** **H**, dzięki temu możemy w prosty sposób prezentować różnego typu komunikaty dla różnego typu błędów. Ważne jest jednak, aby podać instrukcję z dokładnym kodem błędu przed końcową instrukcją **except**, która ma obsłużyć dowolny błąd.

```
Gdy chcesz zakonczyc prace programu wprowadz x jako p
Wprowadz promień okręgu: dwa
Traceback (most recent call last):
  File "C:\Python\Projekt1\pole_kola.py", line 15, in
    policz_pole_okr(promien)
  File "C:\Python\Projekt1\pole_kola.py", line 8, in
    promien2 = float(promien)
ValueError: could not convert string to float: 'dwa'   I

Process finished with exit code 1
```

**4** Jak widać w powyższym przykładzie, wyjątek został poprawnie obsłużony według nazwy. Warto tworzyć obsługę poszczególnych wyjątków, gdyż pozwala to później radzić sobie z błędami nowego typu na bieżąco.

Wróćmy teraz do naszego przykładu z obliczaniem pola powierzchni koła. Znając zasadę działania bloków **try** oraz **except**, możemy obsłużyć wyjątki, o których wiemy, że mogą wystąpić, oraz te, których nie planujemy.

**1** Błąd, jaki pojawia się po wpisaniu ciągu znaków zamiast wartości liczbowej lub znaku innego niż **x**, to **ValueError** **I** – musimy zająć się jego obsługą.

**2** W tym przypadku należy dodać blok **try** oraz **except** wewnątrz pętli **while**, gdyż na tym etapie może pojawić się oczekiwany przez nas błąd. Następnie po instrukcji **except** podajemy nazwę kodową błędu **J** i przykładową informację, która zostanie wyświetlona użytkownikowi.

**3** Dzięki umieszczeniu bloków **try** oraz **except** wewnątrz pętli nieskończonej **while** możemy w prosty sposób wymusić na użytkowniku podanie poprawnych danych. Teraz nasza pętla nawet przy

```
while True:
    try:
        promien = input("Wprowadz promień okręgu: ")
        policz_pole_okr(promien)
    except ValueError:
        print("Podano błędna wartość należy podać liczbę!"   J
              "Podaj wartość raz jeszcze!")
```

```
C:\Python\Projekt1\venv\Scripts\python.exe C:/Python/Projekt1/pole_kola.py
Gdy chcesz zakończyc prace programu wprowadź X jako promień
Wprowadz promień okręgu: 5
Pole powierzchni wynosi:  78.53981633974483
Wprowadz promień okręgu: dwa
Podano błędna wartość należy podać liczbę! Podaj wartość raz jeszcze! K
Wprowadz promień okręgu: 1
Pole powierzchni wynosi:  3.141592653589793
```

tym krytycznym błędzie dalej będzie działać do momentu podania przez użytkownika poprawnych danych **K**.

Poprawienie błędu, który pozwala na obliczanie pola koła z ujemnej wartości promienia, musi być rozwiązane przez wprowadzenie poprawek bezpośrednio do kodu i zastosowanie odpowiednich instrukcji warunkowych. W naszym przypadku, ponieważ całość skryptu jest realizowana wewnątrz pętli, która wywołuje funkcję **policz_pole_okr**, musimy wprowadzić dodatkowy warunek wewnątrz tej właśnie funkcji.
W tym konkretnym przykładzie **L** wystarczyło, że sprawdzamy, czy wartość liczbowa promienia jest większa

```python
# Skrypt do obliczania pola powierzchni koła  L
import math

def policz_pole_okr(promien):
    if promien == "x":
        exit()
    else:
        promien2 = float(promien)
        if promien2 < 0:
            print("Prosze podać dodatnią wartość dla promienia")
        else:
            pole = math.pi*promien2*promien2
            print("Pole powierzchni wynosi: ", pole)
```

```
C:\Python\Projekt1\venv\Scripts\python.exe C:/Python/Projekt1/pole_kola.py
Gdy chcesz zakończyc prace programu wprowadź X jako promień
Wprowadz promień okręgu: -5
Prosze podać dodatnią wartość dla promienia
Wprowadz promień okręgu: |
```

## WARTO WIEDZIEĆ!

Możemy również korzystać z klauzuli **else** przy bloku **except**. W przypadku gdy określimy poprzez użycie bloku **except** obsługę kodów błędu, a żaden błąd nie wystąpi, możemy zrealizować konkretny kawałek kodu. Oznacza to, że możemy w blokach **try** i **except** umieścić działanie konkretnej funkcji lub kawałka kodu, a jeśli nie pojawi się w nim błąd, możemy przejść do realizowania dalszego scenariusza.

```python
try:
    a = 3
    b = 2
    c = a/b
except ZeroDivisionError:
    print("Błąd dzielenia przez zero")
else:
    print("Nie było błędu")
```

```
C:\Python\Projekt1\venv\Scripts\python.exe C:/Python/Pr
Nie było błędu

Process finished with exit code 0
```

od zera. Jeśli tak – to przechodzimy do obliczania pola, jeżeli nie – informujemy użytkownika o błędnie podanej wartości, w następnej iteracji pętli będzie on mógł podać poprawną wartość.

## Bloki try i finally

Koncepcja działania bloku **try** jest zawsze taka sama – wykorzystywany jest on do testowania kawałków kodu w poszukiwaniu błędów. Natomiast **finally** służy zarówno do obsługi błędów, jak i wykonania kawałka kodu, jeśli kod wewnątrz bloku **try** nie spowodował problemów. Jest to połączenie **else** oraz **except**. Instrukcja wewnątrz tego bloku zostanie wykonana zarówno gdy wystąpi błąd, jak i wtedy, gdy błąd się nie pojawi.

```
try:   A
    a = 3
    b = 2
    c = a/b
    print(c)
except:
    print("Błąd dzielenia przez zero")
```

```
C:\Python\Projekt1\venv\Scripts\python.exe C:/Python/
1.5

Process finished with exit code 0
```

W przypadku zastosowania bloków **try** oraz **except** zostanie wykonany tylko i wyłącznie kod zawarty wewnątrz bloku **try** **A** i to on będzie analizowany pod kątem błędów.

**UWAGA!**

**W połączeniu możemy stosować jedynie bloki try oraz except lub bloki try oraz finally. Nie wolno w jednej kombinacji połączyć ich wszystkich jednocześnie. Możemy natomiast zagnieżdżać kolejne kombinacje bloków try oraz except lub finally. Właśnie ten sposób jest najbardziej powszechny dla bloku finnaly.**

eśli błędy nie wystąpią, instrukcja z bloku **except** nie zostanie zrealizowana.

```
try:
    a = 3
    b = 2
    c = a/b
    print(c)
finally:   B
    print("Był błąd lub go nie było")
```

```
C:\Python\Projekt1\venv\Scripts\python.exe C:/Python/Pr
1.5
Był błąd lub go nie było

Process finished with exit code 0
```

Natomiast w przypadku bloku **finally** **B**, zarówno gdy zostanie wygenerowany błąd, jak i gdy błędów nie będzie, instrukcja będzie wykonana.

W tym przykładzie **C** zastosowaliśmy zagnieżdżone bloki **try**. W zewnętrznym bloku, korzystając z instrukcji **except**, wyłapujemy wszystkie możliwe błędy. Natomiast wewnętrzna instrukcja zostanie wykonana zawsze, niezależnie od możliwych błędów. Jest to ważne przy pracy z plikami. Na przykład zamknięcie pliku oznacza, jak już wiemy, zapisanie zmian w pliku. W tym konkretnym przykładzie zostanie zrealizowane niezależnie od tego, czy pojawi się błąd, czy też nie.

```
try:   C
    f = open('testfile','w')
    try:
        f.write('To nasz plik')
    finally:
        print('Zamykam plik')
        f.close()
except:
    print("Błąd związany z plikiem")
```

```
C:\Python\Projekt1\venv\Scripts\python.exe C:/Pytho
Zamykam plik
```

# Debugowanie

Debugowanie to jeden z najważniejszych procesów podczas tworzenia oprogramowania. To nieuniknione, że podczas pisania kodu pojawiają się w nim błędy lub problemy, których pochodzenia nie jesteśmy w stanie ustalić, tylko patrząc na kod. Dlatego też musimy zapoznać się z debugowaniem, czyli procesem usuwania błędów. Uczymy się na błędach. Nauka polega na wyciąganiu wniosków z popełnianych błędów – dzięki temu jesteśmy w stanie szybciej i lepiej zapoznać się z danym tematem.

```
pole_kola ×   kalkulator ×
C:\Python\Projekt1\venv\Scripts\python.exe C:/Python/Projekt1/kalku
1, Dodawanie
2, Odejmowanie
3, Mnożenie
4, Dzielenie
5, Wyjście
Podaj numer opcji: Abc
Traceback (most recent call last):
  File "C:\Python\Projekt1\kalkulator.py", line 22, in <module>
    wybor = int(input("Podaj numer opcji: "))
A ValueError: invalid literal for int() with base 10: 'Abc'
```

Pierwsza styczność z błędami krytycznymi w Pythonie może dla początkujących być nieco onieśmielająca. Pojawiają się skomplikowane wyrażenia, specjalne kody błędów, informacje o wierszu wystąpienia błędu itp. Wszystko to jednak ma na celu doprowadzenie nas do rozwiązania problemów z naszym kodem.

Na poprzednich stronach częściowo zapoznawaliśmy się z prezentacją błędów w Pythonie – teraz podsumujemy przedstawione już przykłady i dodatkowo skorzystamy z interaktywnego debuggera IDE PyCharm.

Do tej pory za każdym razem, gdy nasz skrypt nie zadziałał prawidłowo, w konsoli były prezentowane wiersze oznaczone czerwonym kolorem z opisem, co poszło nie tak. Warto nauczyć się odpowiednio odczytywać te komunikaty, gdyż czasem nie jest konieczne pełne debugowanie, a jedynie zrozumienie i odpowiednie zinterpretowanie informacji o błędach.

W skrypcie naszego kalkulatora po wprowadzeniu niepoprawnych danych wejściowych pojawia się błąd **ValueError** **A** – nazwa błędu znajduje się zawsze w ostatniej linii, dodatkowo załączany jest krótki opis, co dokładnie oznacza w tym przypadku błąd oraz przez jaką wartość został spowodowany. W tym przypadku jest to podanie wartości innej niż liczbowa z systemu dziesiętnego.

Powyżej znajduje się **Traceback**, czyli historia operacji, które spowodowały pojawienie się błędu. W tym konkretnie przypadku miało to miejsce w naszym skrypcie dokładnie w wierszu **22**.

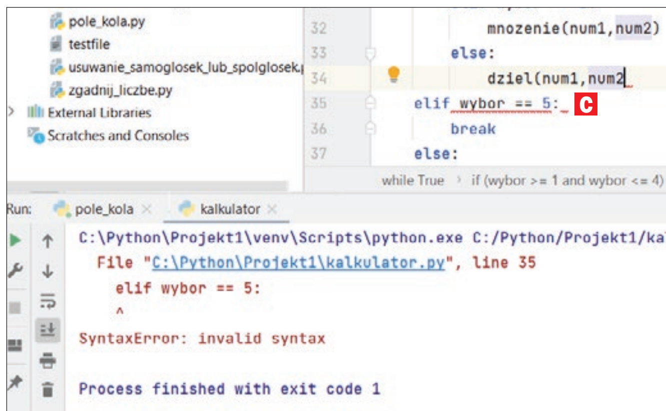
```
   19     print("5, Wyjście")
   20
   21     while True:
B  22         wybor = int(input("Podaj numer opcji: "))
   23         if (wybor >= 1 and wybor <= 4):
   24             print("Wprowadz dwie liczby")
```

W PyCharm po lewej stronie kodu znajdują się numery wierszy. Dzięki temu szybko zlokalizujemy wiersz, w którym wystąpił błąd. W tym konkretnym przypadku w wierszu **22** **B** dokonujemy bezpośredniej konwersji danych, które wprowadza użytkownik poprzez funkcje **input** w domyślnym typie **string** (ciąg znaków) do typu **int**, czyli liczb całkowitych.

Podanie ciągu znaków spowodowało pojawienie się błędu.

Tego typu błędy omówiliśmy już po części w ramach bloków **try** oraz **except**. Najczę-

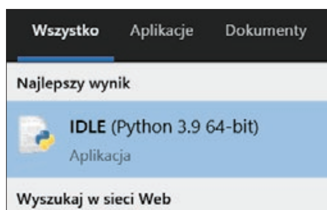
ściej jednak w trakcie tworzenia skryptu programiści zapominają o wstawieniu odpowiedniej liczby nawiasów, wcięć, cudzysłowów i różnego typu innych znaków, które są kluczowe dla składni i dla poprawnego działania programu.
Najczęściej w przypadku błędu programisty dochodzi do tego, że zostaje wygenerowany błąd typu **SyntaxError**, czyli błąd składni. W naszym przykładzie otrzymaliśmy informację o błędzie w wierszu **35** **C**. Tak naprawdę to brak nawiasu w wierszu **34** sprawia, że błąd jest generowany. Jeśli pracujemy w PyCharm, takie błędy są dość proste do wyłapania, gdyż program sam zaznacza błędne wyrażenia, podkreślając je na czerwono.



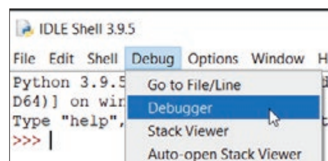

## Debugger Pythona

Python nie tylko informuje nas w przejrzysty sposób o błędach, ale ma również wbudowany **debugger**, czyli narzędzie, które pomaga w rozwiązywaniu problemów i usuwaniu błędów. Jest to moduł **pdb**. Obsługa tego modułu jest nawet wbudowana w IDLE, jest on też wspierany przez PyCharm.
Jeśli chcemy uruchomić nasz skrypt w trybie debugowania, należy w konsoli wpisać polecenie **python -m pdb program.py**. Możemy również zrobić to z poziomu IDLE, gdzie można nawet aktywować debugger dla sesji interaktywnej.

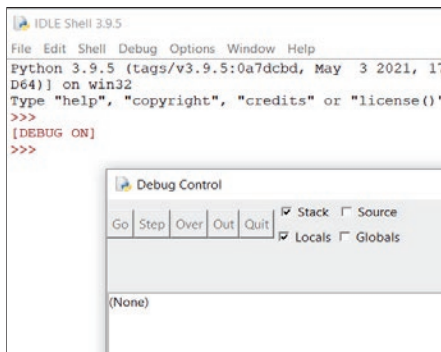
**1** Uruchamiamy IDLE, wyszukując program w pasku wyszukiwania Windows.




**2** Następnie w oknie programu IDLE klikamy na górnym pasku na **Debug**, a potem na **Debugger**.

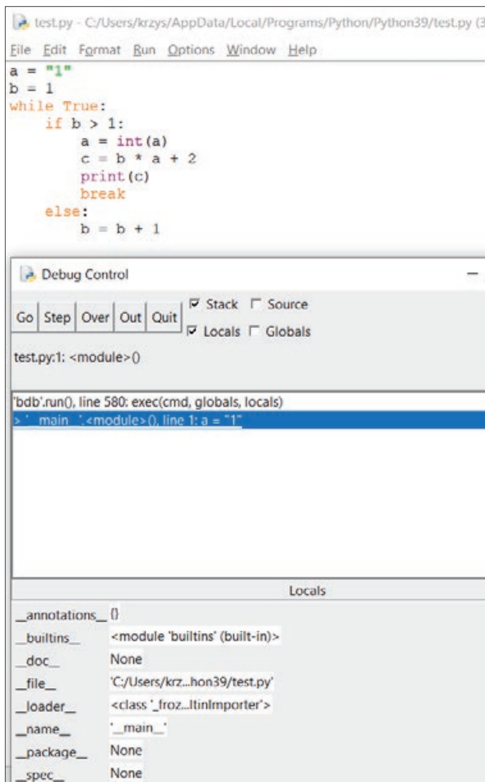



**3** Pojawi się nowe okno **Debug Control**, w którym będziemy mieć podgląd od wewnątrz na działanie naszego kodu. W samej konsoli IDLE pojawi się również napis **DEBUG ON**, co oznacza, że aktywny jest tryb debugowania.

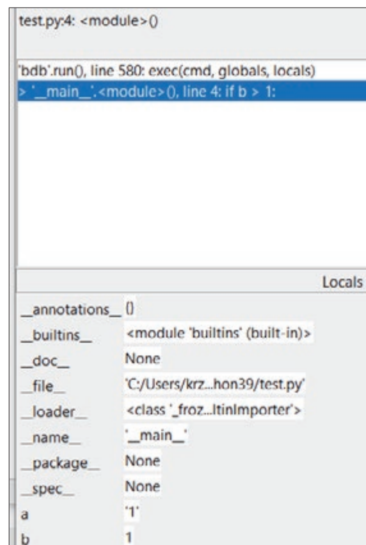



**4** Następnie możemy wprowadzić kawałek kodu, który chcemy debugować. Proces ten nie zawsze ma na celu tylko i wyłącznie

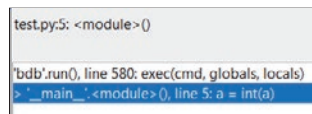
znajdowanie i usuwanie błędów. Czasami jest przydatny w lepszym zrozumieniu działania kodu programu, gdyż pozwala programiście krok po kroku sprawdzać, co dzieje się z poszczególnymi zmiennymi oraz jakie dokładnie operacje są wykonywane.




**5** Możemy obsługiwać debugger, korzystając z kilku przycisków – ich dokładne działanie poznamy w dalszej części książki. Do podstawowego zastosowania wystarczy nam przycisk **Step**, który pozwala wykonywać skrypt wiersz po wierszu. Już po trzech pierwszych krokach debugger u dołu okna wyświetla informacje na temat zarejestrowanych zmiennych oraz przypisanych im wartości. W środkowym oknie widzimy, na jakim kroku aktualnie jesteśmy – ponieważ warunek logiczny nie został spełniony, trafiliśmy do instrukcji **else**.




**6** W jednym z kolejnych kroków dojdziemy do konwersji zmiennej **a** z ciągu znaków




na wartość liczbową. Wtedy w dolnej części okna debuggera będziemy mogli zaobserwować zmianę.

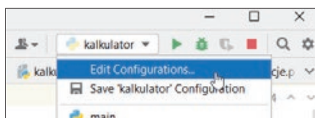
| | |
|---|---|
| __package__ | None |
| __spec__ | None |
| a | 1 |
| b | 2 |

## Debugowanie w PyCharm

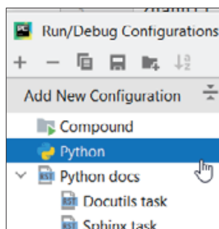
Proces debugowania w PyCharm jest zdecydowanie bardziej wygodny i prosty. Dzięki wbudowanym narzędziom możemy skonfigurować osobne środowiska dla każdego projektu i testować zachowanie naszego programu w różnych, z góry określonych warunkach, na przykład w starszej wersji Pythona lub z najnowszymi wersjami bibliotek.

**1** Zaczynamy od dodania środowiska debugowania do naszego projektu – jeśli wykonywaliśmy poprawnie wszystkie kroki dotyczące instalacji PyCharm i tworzenia projektu, powinno być ono automatycznie utworzone. Jeśli nie, klikamy na górnym pas-

ku programu na nazwę skryptu, a następnie na **Edit Configurations**.



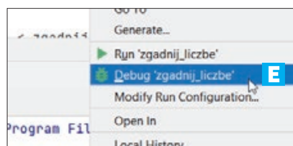

**2** Następnie po lewej stronie klikamy na plus w celu dodania nowej konfiguracji, a później na **Python**.



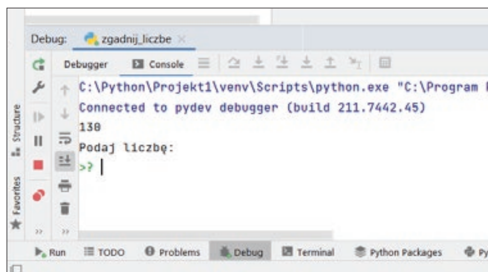

**3** Teraz dodajemy nową nazwę **A**, wskazujemy lokalizację skryptu **B**, wybieramy interpreter **C** i klikamy na **OK**.

**4** Od tej pory, aby rozpocząć debugowanie skryptu, wystarczy kliknąć na ikonę na górnym pasku **D** lub kliknąć prawym przyciskiem myszy w polu skryptu i wybrać z menu dialogowego opcję **Debug** **E**.

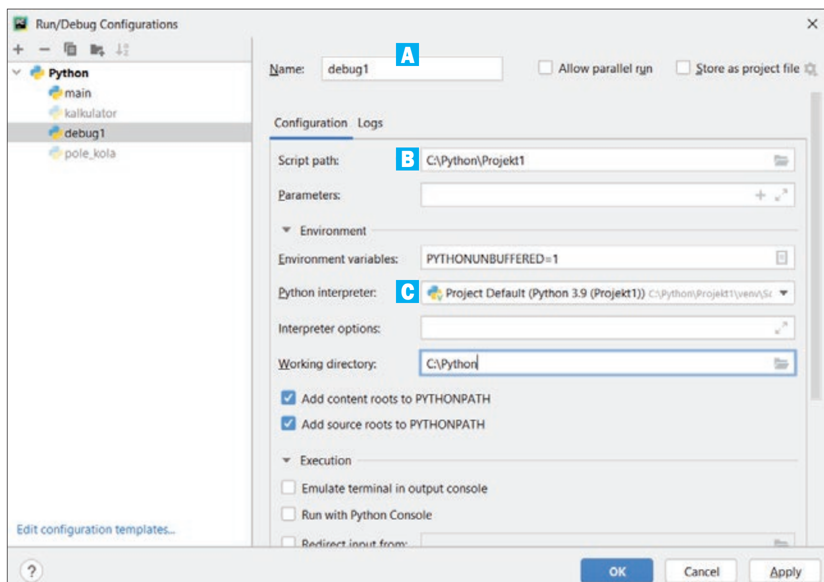



**5** Teraz w dolnej części interfejsu pojawi się okno **Debug** z dwoma zakładkami – **Debugger** i **Console**, w tym drugim mamy możliwość interakcji z programem, w tym pierwszym z kolei mamy pełny podgląd na działanie skryptu.



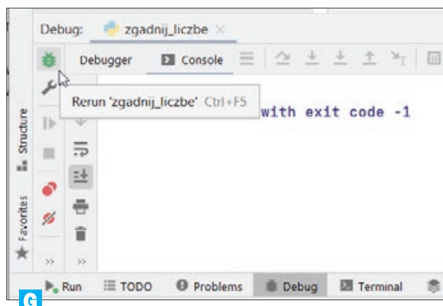

**6** Od razu nie mamy jednak możliwości sprawdzania krok po kroku działań naszego skryptu, w tym celu musimy ustawić **breakpoint**, czyli punkt zatrzymania. Wyobraźmy



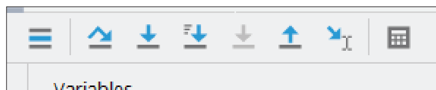
sobie sytuację, że nasz skrypt ma 3000 wierszy, a interesuje nas sprawdzenie poprawności działania kodu w liniach 555 i 600, wtedy ustawiamy **breakpoint** na na przykład wiersz 553 i od tego miejsca swoją pracę rozpocznie **Debugger**. Utworzenie breakpointu polega na kliknięciu na wolną przestrzeń obok numeru wiersza – pojawi się tam wtedy symbol czerwonej kropki **F**. **Uwaga!** Nie możemy ustalić breakpointu w pustym wierszu.




**7** Po ustaleniu punktu stopu (breakpointu) ponownie uruchamiamy skrypt w trybie debugowania. Możemy to zrobić, klikając na zieloną ikonę **G** w oknie **Debug** u dołu interfejsu.

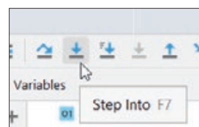



**9** Do obsługi posłużą nam przyciski na górnym pasku debuggera, podobnie jak w przypadku PDG mamy do dyspozycji opcje: **step over**, **step into**, **step out** i kilka innych.

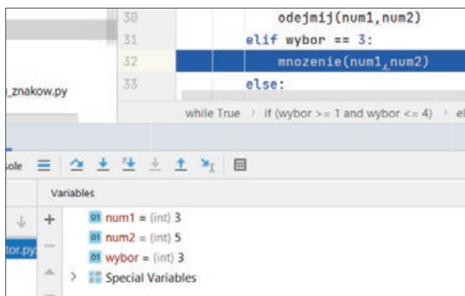



**10** W celu przechodzenia krok po kroku należy kliknąć na **step over**, w celu wejścia wewnątrz funkcji klikamy na **step into**, w celu opuszczenia debugowania klikamy na **step out** lub jeśli jesteśmy wewnątrz funkcji, aby wyjść z samej funkcji.

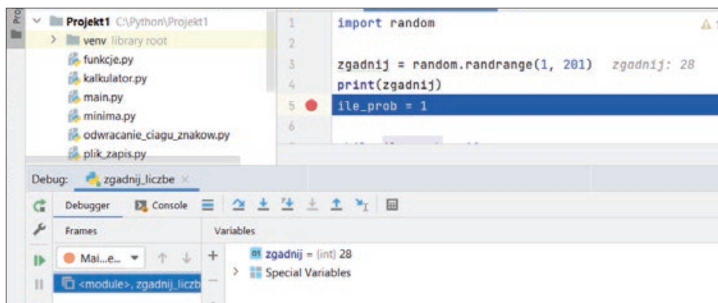
## Wchodzimy wewnątrz funkcji i ją opuszczamy




Tak jak już wcześniej napisaliśmy, klikając na przycisk **Step into**, możemy przejść do wnętrza konkretnej funkcji. Jeśli tego nie zrobimy, domyślnie debugger wykona funkcję, a my będziemy mogli poruszać się dalej po kodzie krok po kroku do przodu. Jeżeli więc problem może



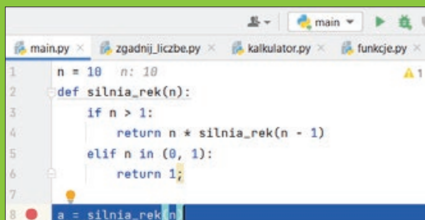

**8** Teraz w oknie z naszym kodem będą pojawiać się wartości zmiennych, a dodatkowo w oknie debuggera poznamy nawet typ przypisany do zmiennej.



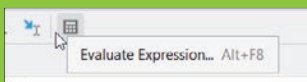
# OKNO WYKONYWANIA KODU

Dodatkowo w PyCharm mamy do dyspozycji narzędzie **Evaluate expression**, służy ono do dynamicznego testowania kodu, funkcji, wyrażeń. Dzięki niemu w trakcie pracy debuggera po zatrzymaniu wykonywania kodu możemy testować różne działania wewnątrz naszego skryptu. Bardzo często tego typu testy pozwalają na wczesnym etapie wykryć nieprawidłowości wewnątrz kodu.

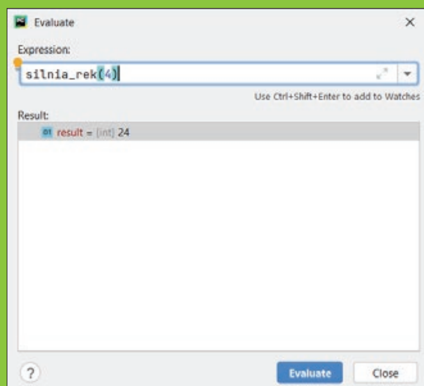
**1** Ustawiamy punkt stopu wewnątrz naszego kodu, następnie uruchamiamy go w trybie debugowania.



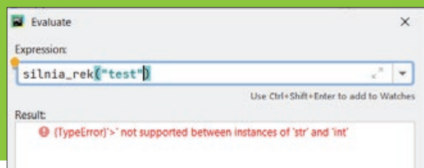

**2** Teraz możemy kliknąć na przycisk **Evaluate expression**.



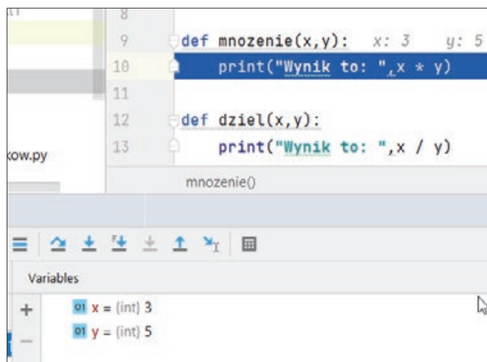

**3** Pojawi się wtedy okno **Evaluate**, w polu **Expression** podajemy wyrażenie, które chcemy przetestować. Może być to jedna z funkcji, może być to dowolne działanie niezwiązane z samym skryptem, oparte na zmiennych wewnątrz skryptu, i wiele więcej. Możemy na przykład wywołać funkcję z podanymi ręcznie przez nas argumentami.




**4** Korzystając z tej metody, możemy sprawdzić, w jakich sytuacjach wystąpi błąd dla danej funkcji lub wyrażenia i wtedy odpowiednio poprzez obsługę wyjątków zabezpieczyć nasz kod.

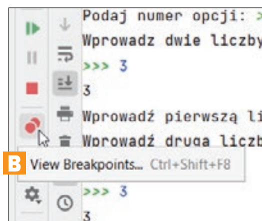


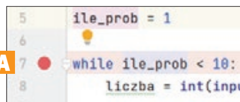

występować wewnątrz funkcji lub chcemy prześledzić jej działanie w momencie, gdy podświetlony w debuggerze zostanie wiersz z funkcją, musimy kliknąć na **Step into**. Wewnątrz funkcji również możemy podglądać wartości jej lokalnych zmiennych i poruszać się krok po kroku. W celu opuszczenia funkcji i powrotu do głównej części kodu możemy albo „przeklikać się" do samego końca wiersz po wierszu lub kliknąć na **Step out**, aby opuścić funkcję od razu. Jeśli jest to bardzo rozbudowana funkcja, może nam to zaoszczędzić mnóstwo czasu.



## Warunkowe punkty stopu

Wiemy już, jak korzystać z breakpointów. Warto dodatkowo poznać możliwości warunkowych punktów stopu. Pozwalają one na uruchomienie debuggera tylko w przypadku, gdy jakiś warunek wstępny zostanie spełniony. Załóżmy, że pracujemy nad skryptem, który sortuje pewne dane i po kilkudziesięciu iteracjach zawsze przestaje działać poprawnie. Wtedy, jeśli ustawimy warunkowy breakpoint na przykład na pięćdziesiątą iterację pętli odpowiedzialnej za sortowanie – znacząco szybciej dojdziemy do etapu, gdzie występuje błąd.



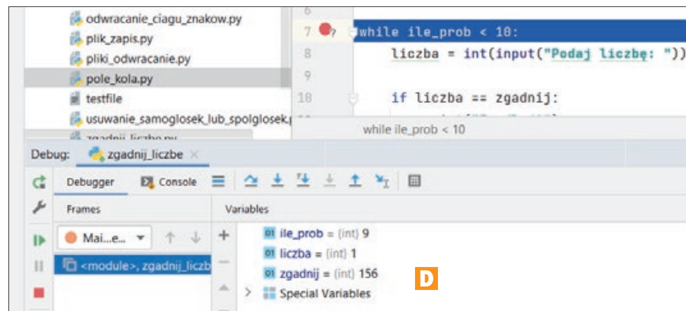



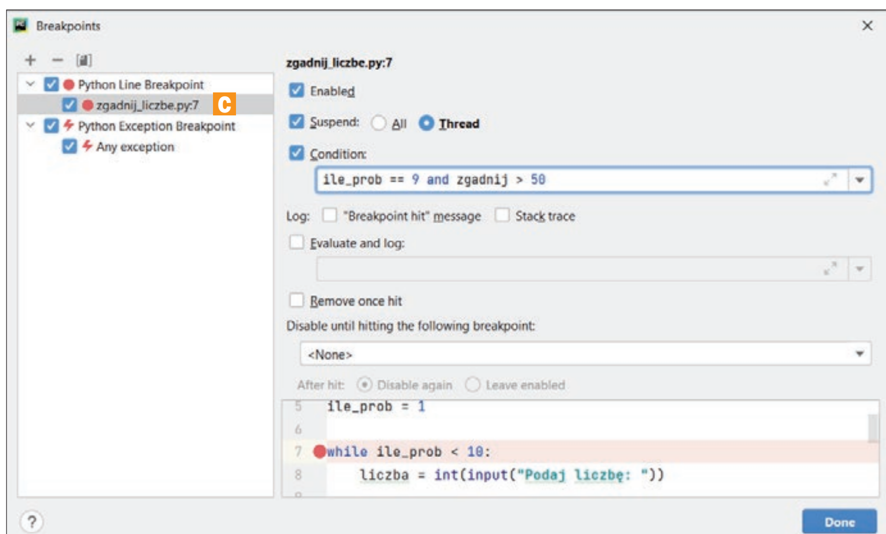

**1** Dodajemy punkt stopu w odpowiadającym nam miejscu wewnątrz skryptu **A**.

**2** Teraz wciskamy kombinację klawiszy ctrl+shift+F8, w oknie **Debugger** po lewej stronie klikamy na **View Breakpoints** **B**.

**3** W nowym oknie, po lewej stronie, wybieramy nasz breakpoint **C**. Po prawej stronie zaznaczamy opcję **Condition** i w pustym polu tekstowym wprowadzamy nasz warunek, może być on dowolnie rozbudowany. Musimy jednak pamiętać, aby stosować się do składni Pythona. Na koniec klikamy na **Done**.

**4** Teraz po uruchomieniu procesu debugowania wykonywanie skryptu zostanie zatrzymane na dziewiątej iteracji, gdy dodatkowo wylosowana wartość będzie większa niż 50 **D**.

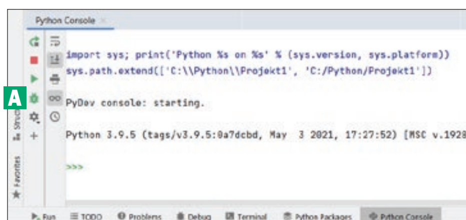


## Interaktywne okno sesji Pythona

W procesie debugowania czasem może zdarzyć się, że winę za błędne działanie skryptu ponosi jakaś biblioteka lub moduł zewnętrzny, który importujemy. Warto wtedy skorzystać z dostępnej w PyCharm interaktywnej sesji Pythona. W tym oknie będziemy mogli bardzo szybko zweryfikować poprawność działania pojedynczych linijek kodu.

**1** W celu dołączenia do interaktywnej sesji debuggera klikamy w zakładce **Python Console** po lewej stronie na zieloną ikonę debuggera **A**.




**2** Powinna pojawić się informacja **Debugger connected** **B**.

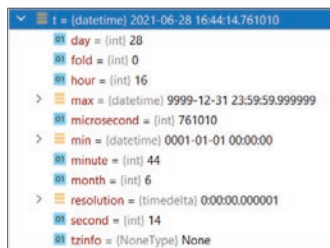
**3** Od tej chwili nasza interaktywna sesja korzysta z możliwości debuggera i możemy testować różne linie kodu, które sprawiają nam problemy.

W każdej chwili w trakcie pracy w interaktywnej sesji z podłączonym debuggerem na bieżąco będziemy mieć podgląd na wartości zmiennych oraz na ich typy w oknie po prawej stronie **C**.

## Sprawdzamy szczegóły w oknie debugowania

Do tej pory skupiliśmy się jedynie na wartościach i typach zmiennych, jakie podpowiada nam debugger. Możemy wejść nieco głębiej i sprawdzić szczegóły.

**1** Po utworzeniu obiektu klasy **datetime** w debuggerze możemy rozwinąć listę wartości tego obiektu.

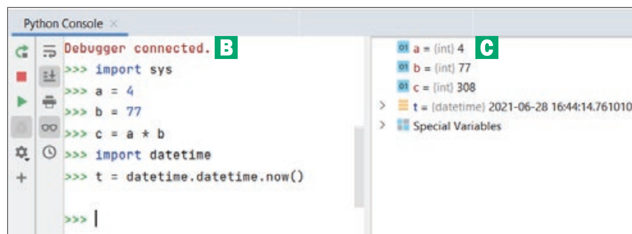


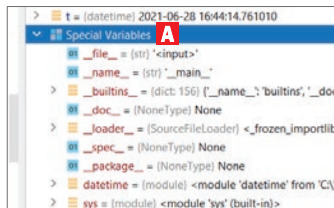

**2** Odwołując się bezpośrednio do samego obiektu **t**, uzyskamy dane ogólne. Na liście możemy zapoznać się z poszczególnymi polami tego obiektu, jak na przykład pole **day**, a następnie możemy spróbować się do nich odwołać.




**3** Dodatkowo, rozwijając listę **Special Variables** **A**, uzyskamy dostęp do danych na temat załadowanych klas, modułów, konstruktorów itp. Ostatnio dodane moduły są zawsze u dołu listy – w naszym przypadku jest to moduł **datetime** oraz **sys**.

Jeśli nie znamy dokładnie wszystkich pól jakiejś klasy, dobrym pomysłem jest utworzenie dla niej nowego obiektu, a następnie sprawdzenie go w debuggerze. W ten sposób szybko dowiemy się, jakie pola zostają przypisane do obiektu oraz jakie są ich wartości. Ta wiedza pomoże nam w dalszej pracy z różnego typu obiektami.



# 6 Programowanie obiektowe

**Python jest językiem typowo obiektowym. Jak już zdążyliśmy zauważyć w przykładach z datą i czasem, korzystając z obiektów tworzonych w klasach, możemy przypisywać im dodatkowe atrybuty. Teraz dowiemy się, czym są dokładnie klasy i do czego przydaje się obiektowe programowanie**

## Klasy

Opisywane do tej pory skrypty były dość proste i nie wymagały od nas wprowadzania dużej ilości kodu.
W praktyce jednak aplikacje są zbudowane z setek, jak nie tysięcy linijek kodu i mają mnóstwo różnego rodzaju powiązań. Zaczyna pojawiać się problem ze zmiennymi, funkcjami itp. Po dłuższym czasie nawet sam autor programu może zgubić się w kodzie i zapomnieć, czy już tworzył rozwiązanie do danego problemu.
Części kodu mają cechy wspólne, które pokrywają się w pewnych aspektach, a w innych się różnią. Jak napisać czysty kod bez zbędnych powtórzeń?
Rozwiązaniem tego problemu jest programowanie obiektowe, czyli takie, w którym definiujemy klasy.
Klasa i obiekt to dwa różne, choć zbliżone do siebie rozwiązania. Programowanie obiektowe polega na powiązaniu danych z czynnościami, jakie na tych danych można wykonać. Dla przykładu – każdy człowiek ma imię, nazwisko, wiek. Są to tylko i wyłącznie dane. Pytanie więc brzmi: Jakie czynności możemy

### POLA A METODY

Zanim przejdziemy do tworzenia naszej pierwszej klasy, musimy jeszcze omówić koncepcję pól i metod. Z obydwoma się już spotkaliśmy. Wszystkie dane przedstawiamy za pomocą pól, polem może być na przykład pojedyncza zmienna przechowująca wartość liczbową. W naszym przykładzie związanym z człowiekiem (opisany na tej i następnych stronach) powinniśmy więc mieć trzy pola: imię, nazwisko, wiek. Każdy obiekt naszej klasy będzie miał własne trzy pola, więc im więcej obiektów, tym więcej pól.
Metoda to natomiast czynność związana z danym polem. Inaczej mówiąc, jest to funkcja. A dokładniej mówiąc, to funkcja bezpośrednio powiązana z daną klasą. Mówimy, że metoda jest wywoływana na potrzeby konkretnego obiektu. Sam dostęp do pól klasy, odczytywanie, modyfikowanie sprawia, że kawałek kodu jest metodą danej klasy.

powiązać z tymi danymi? Najprostsze czynności to przedstawienie się czy podanie swojego wieku. Wewnątrz Pythona spotkaliśmy do tej pory bardzo podstawowe klasy, jak lista. Na liście możemy umieścić i posortować dane dotyczące kilku osób, czyli zapisać dane i wykonać określone czynności.
Klasa sama w sobie to konstrukcja abstrakcyjna, z powyższego przykładu wynika, że może przechować dane dotyczące imienia, nazwiska oraz wieku człowieka i nic więcej. Klasa nie pokazuje, jakie konkretnie czynności są związane z tymi danymi.
Obiekt natomiast jest już konkretnym, jasno określonym, „powołanym do życia" ucieleśnieniem tej klasy i zajmuje fizycznie pamięć. Obiektów jednej klasy może być praktycznie nieskończenie wiele, możemy również odwoływać się do poszczególnych obiektów, gdyż mają one przypisane do siebie konkretne wartości.

## Tworzymy własną klasę *czlowiek*

Klasę należy zawsze umieszczać w kodzie przed właściwą częścią skryptu. Może być również zapisana jako moduł i zaimportowana do programu – dzięki temu nie robimy sobie w kodzie bałaganu.

**1** Zaczynamy od definicji klasy – jest ona dość prosta, wystarczy podać instrukcję **class**, nazwę klasy i znak **:**.

```
main.py ×   zgadnij_liczbe.
class czlowiek:
```

**2** Następnie musimy zdefiniować konstruktor służący do inicjalizacji pól klasy. Konstruktor jest specjalnym rodzajem metody. Metody wewnątrz klasy przyjmują zawsze przynajmniej argument **self** – jest on obiektem, na którego rzecz są wykonywane. Dodatkowo podajemy w konstruktorze trzy pola, które również są jego argumentami. Można powiedzieć, że konstruktor inicjalizujący jest przepisem na konstrukcję tworzonego obiektu. Pozwala on na zainicjalizowanie konkretnych pól na rzecz obiektu. W naszym przykładzie chcemy, aby dane: imię, nazwisko oraz wiek, zostały przypisane do odpowiednich pól w klasie. W dalszej części kodu zobaczymy, dlaczego jest to istotne.

```
class czlowiek:
    def __init__(self, imie, nazwisko, wiek):
        self.imie = imie
        self.nazwisko = nazwisko
        self.wiek = wiek
```

**3** Należy zwrócić uwagę na konstrukcje, jak **self.imie**. Będzie ona często widoczna w kodzie, gdzie występują klasy. Zawsze przed kropką znajduje się zmienna określająca obiekt jakiejś klasy, a po kropce będziemy odwoływać się do jakiejś składowej tego konkretnego obiektu. Wcześniej korzystaliśmy z tego zapisu, na przykład odwołując się do klasy **datetime**, wywoływaliśmy aktualny czas zapisem **datetime.now()**. Python nie wie, jakie pola ma mieć dany obiekt, właśnie w konstruktorze dokonujemy ich definicji, dzięki temu w każdej kolejnej metodzie możemy się do nich odwoływać.

```
def __init__(self, imie
    self.imie = imie
    self.nazwisko = naz
```

**4** Następnie przechodzimy do zdefiniowania metod klasy, czyli czynności powiązanych z naszymi danymi. Wszystkie przykładowe metody nie przyjmują żadnych dodatkowych parametrów oprócz **self** – jest on obowiązkowy. Pierwsze dwie metody nic nie zwracają – nie korzystają z instruk-

```
def jak_sie_nazywa(self):
    print("Nazywam się", self.imie, self.nazwisko)
def ile_lat(self):
    print("Mam", self.wiek, "lat")
def urodziny(self):
    wiek_wczesniej = self.wiek
    self.wiek += 1
    return wiek_wczesniej
```

cji **return**. Służą jedynie do wyświetlenia dla użytkownika przygotowanych wiadomości. Ponieważ metody są wywoływane na rzecz konkretnego obiektu, a wiemy, że tworzone dla tej klasy obiekty będą miały trzy zdefiniowane pola, możemy się do nich bezpośrednio odwoływać w taki sposób, jak zostały określone w konstruktorze. Ostatnia metoda – **urodziny** – zwraca pewną wartość, wewnątrz skryptu możemy ją obsłużyć tak, jak robiliśmy to z wartościami zwracanymi przez funkcje.

**5** Teraz możemy przejść do tworzenia obiektów naszej przykładowej klasy **czlowiek**. Podajemy w tym celu nazwę nowego obiektu i przypisujemy mu klasę z wszystkimi polami poza **self**, gdyż jest ono na potrzeby własne konstruktora.

```
        self.wiek += 1
        return wiek_wczesniej

Jan = czlowiek("Jan", "Kowalski", 55)
Piotr = czlowiek("Piotr","Nowak", 99)
```

**6** Następnie możemy wywołać metody dla naszych dwóch nowych obiektów. Podajemy nazwę obiektu, a po kropce metodę, jaką chcemy wywołać na rzecz danego obiektu.

```
Jan.jak_sie_nazywa()
Jan.ile_lat()
Piotr.jak_sie_nazywa()
Piotr.ile_lat()
```

**7** W przypadku metody **urodziny** **A** zwracana wartość powoduje zmianę

```
Piotr.urodziny() A
print(Piotr.imie,"miał urodziny!")
Piotr.jak_sie_nazywa()
Piotr.ile_lat()
```

przypisanego przez konstruktor parametru, więc po jej wywołaniu możemy jeszcze raz wyświetlić metodę **ile_lat** **B** i będzie ona zawierała nowe dane.

```
pole_kola × | main ×
C:\Python\Projekt1\venv\Scripts\python.exe C:/
Nazywam się Jan Kowalski                    B
Mam 55 lat
Nazywam się Piotr Nowak
Mam 99 lat
Piotr miał urodziny!
Nazywam się Piotr Nowak
Mam 100 lat

Process finished with exit code 0
```

## Widoczność składowych klasy

Tworząc klasę, nie zawsze będziemy chcieli, aby każda osoba miała dowolny dostęp do składowych klasy. Czasem lepiej, aby niektóre pola, a nawet metody pozostały ukryte. W większości języków obiektowych wyróżniamy trzy klasy dostępności:

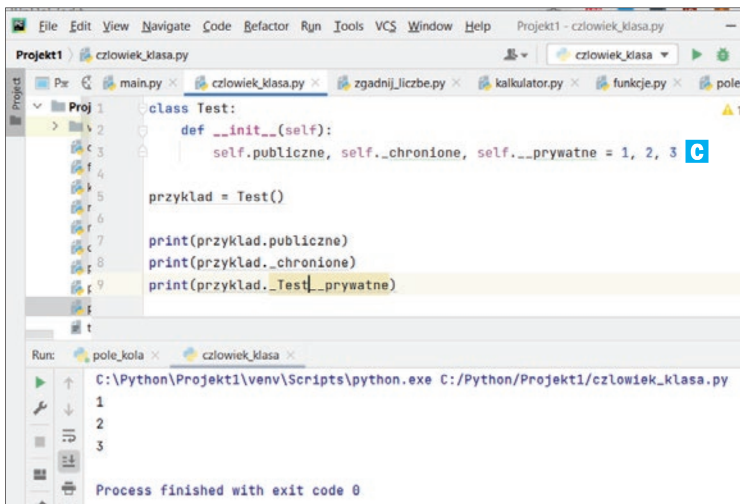
- **Publiczne** – mają do nich dostęp wszyscy
- **Chronione** – mają dostęp jedynie klasy dziedziczące
- **Prywatne** – dostęp ma jedynie jedna konkretna klasa

W Pythonie natomiast jest nieco inaczej. Nie da się w rzeczywistości niczego ukryć, tworząc klasę. Wszystkie elementy dotyczące klasy do tej pory nasze IDE w trakcie kodowania nam podpowiadało – gdy wpisujemy po wcześniejszym utworzeniu obiektu jego nazwę oraz kropkę, pojawia się dużo podpowiedzi dotyczących zarówno pól obiektu, jak i metod.

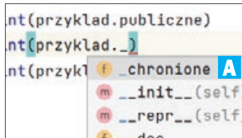
```
Jan.
  f wiek                       czlowiek
  f imie                       czlowiek
  m ile_lat(self)              czlowiek
  m jak_sie_nazywa(self)       czlowiek
  f nazwisko                   czlowiek
```

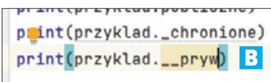
W Pythonie wprowadzono rozwiązanie, które pozwala na teoretyczne ukrycie pól i metod poprzez odpowiednie ich nazwanie. Nazwy bez żadnego poprzedzającego je podkreślenia może traktować jako publicz-

```
print(przyklad.publiczne)
print(przyklad.)
print(przykl  f publiczne
              m __init__(self)
                par
```




ne, z jednym podkreśleniem – jako chronione, czyli nie będą one domyślnie na liście podpowiadanych nazw. Jeśli jednak wpiszemy znak podkreślenia, pojawi się „chronione” pole na liście podpowiedzi **A**.

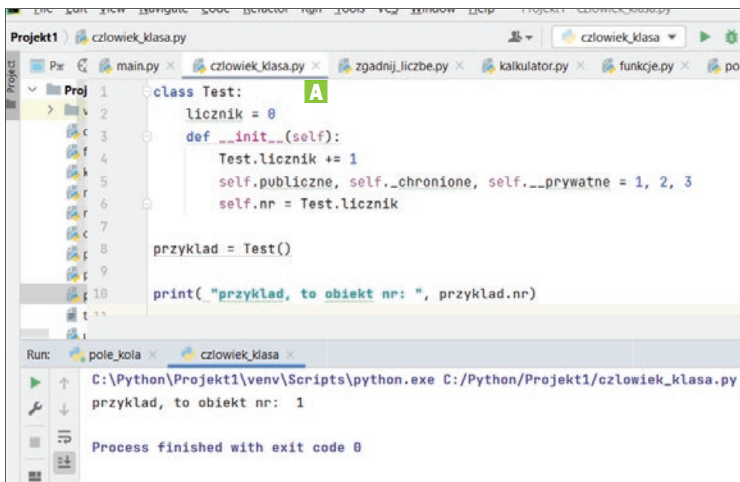



Pole (lub metoda) teoretycznie prywatne **B** nawet po podaniu pełnej nazwy nie znajdzie się na liście podpowiedzi. Aby móc z niego skorzystać, musimy poprzedzić jego nazwę podkreśleniem i nazwą klasy – wtedy uzyskamy do niego dostęp **C**.

## Metody i pola statyczne

W poprzednich przykładach opisaliśmy klasy, które wymagały utworzenia konkretnego obiektu, aby odwołać się do ich pól i metod. Czasem jednak zachodzi potrzeba, aby utworzyć unikalną zmienną dla całej klasy, do której będzie można odwołać się w dowolnym momencie i nie będzie ona przypisana do obiektu, tylko raczej do samej klasy. Może być to również metoda klasy.



W tym przykładzie A statycznym polem jest **licznik**, który wywoływany jest w konstruktorze poprzez odwołanie do samej klasy, a następnie nazwy pola. W tym konkretnym przykładzie statyczne pole pozwoliło nam na sprawdzenie numeru utworzonego obiektu.

```
przyklad = Test()        B
a = Test.licznik
print(a)
print( "przyklad, to obiek
```

```
pole_kola    czlowiek_klasa
C:\Python\Projekt1\venv\Scripts\py
1
przyklad, to obiekt nr:  1
```

Możemy odwoływać się bezpośrednio do pól statycznych klasy wewnątrz głównego kodu. Wystarczy wywołać klasę B, a następnie jej pole lub metodę statyczną.

## Dziedziczenie klas

Pomimo skomplikowanej nazwy jest to dość prosty proces, który – jak sama nazwa wskazuje – polega na dziedziczeniu. W Pythonie dziedziczenie w odniesieniu do klas polega na przekazywaniu do kolejnej podklasy pewnych cech klasy nadrzędnej.

Za przykład posłużą nam ptaki. Możemy dla nich utworzyć klasę, której wspólnymi cechami mogą być skrzydła, dzioby, kolor itp. Ptaki mogą również wykonywać podobne czynności: latać, jeść, wydawać dźwięki.

W celu opisania trzech ptaków moglibyśmy utworzyć trzy osobne klasy, na przykład orzeł, sowa, czapla, i nadać im pola, konstruktory oraz metody. Skoro jednak staramy się wszystko zautomatyzować i zmniejszyć nakład naszej pracy, warto skorzystać z dziedziczenia.

Dla przykładu utworzymy klasę **ptaki**, będzie ona zawierała w sobie wszystkie wspólne cechy przedstawione powyżej. Dodatkowo utworzymy bardzo proste klasy, jak klasa **sowa**, potomek klasy ptaki, z dodatkową cechą: poluje w nocy.

Oczywiście nadal musimy stworzyć konkretne klasy dla poszczególnych gatunków – jednak wszystkie pola wspólne zostaną dokładnie opisane w głównej klasie, do której poprzez dziedziczenie będzie miała dostęp zarówno klasa **sowa**, jak i **orzel** czy **czapla**.

```
class ptak: A
    def __init__(self, gatunek, szybkosc):
        self.gatunek = gatunek
        self.szybkosc = szybkosc
    def lec(self):
        print("Tu", self.gatunek,
              ". Startuje, osiągam maksymalnie", self.szybkosc)
```

```
class orzel(ptak): B
    def polowanie(self):
        print("Tu", self.gatunek, ". Rozpocząłem polowanie")

class czapla(ptak): C
    def polow(self):
        print("Tu", self.gatunek, ". Tu łowię ryby")
    def lec(self):
        print("Tu", self.gatunek, ". Nie lecę, będę jadł")
```

**1** Na początku definiujemy klasę **ptak** A, tworzymy jej konstruktor, przypisujemy pola i tworzymy metody.

**2** Następnie tworzymy kolejne klasy – **orzel** B oraz **czapla** C. Bardzo ważne jest podanie klasy, z której będziemy dziedziczyć w nawiasach po nazwie klasy. Taki zapis oznacza, że klasy te dziedziczą od klasy **ptak**. Każdy obiekt klasy **orzel** i **czapla** ma dostęp do tego samego konstruktora oraz metody **lec**. Dodatkowo klasa **czapla** nadpisała metodę **lec**, ponieważ dla niej aktualna jest inna czynność – nie będzie teraz latać, tylko zamierza jeść.

**3** Możemy teraz przejść do wywołania obiektów. Pamiętajmy, aby najpierw je utworzyć D. Po utworzeniu obiektów zgodnie z konstruktorem głównym możemy dowolnie z nich korzystać.

D

```
ptak_o = orzel("orzeł",33)
ptak_c = czapla("czapla",25)

ptak_o.lec()
ptak_o.polowanie()
ptak_c.lec()
ptak_c.polow()
```

**4** Warto zwrócić uwagę, że ta sama metoda wykonana przez klasę **czapla** oraz **orzel** – **lec** – zwraca różne wartości dla różnych klas.

## Dodajemy konstruktor dla potomka

Powyższy zapis jest poprawny w sensie takim, że kod zadziała i skrypt się wykona, jednak możemy znacznie poprawić czystość naszego kodu. Dla przykładu – pierwszym parametrem, jaki podajemy, jest gatunek czy nazwa ptaka. Jednocześnie dla każdego ptaka tworzymy osobną klasę o dokładnie tej samej nazwie. Możemy wykorzystać do tego celu konstruktor potomka. Do tej pory wykorzystywaliśmy konstruktor tylko i wyłącznie klasy **ptak**, klasy dziedziczące nie miały żadnych własnych konstruktorów.

**1** W naszym przykładzie potomkowie to **orzel** oraz **czapla**. Dodajemy konstruktor na początku każdej z tych klas A.

**2** Zwróćmy uwagę na to, co zostało dodane do poszczególnych klas. W klasie **ptak**

```

class ptak:
    def __init__(self, gatunek, szybkosc):
        self.gatunek = gatunek
        self.szybkosc = szybkosc
    def lec(self):
        print("Tu", self.gatunek,
              ". Startuje, osiągam maksymalnie", self.szybkosc)

class orzel(ptak):
    def __init__(self, szybkosc):
        super().__init__("orzel", szybkosc)
    def polowanie(self):
        print("Tu", self.gatunek, ". Rozpocząłem polowanie")

class czapla(ptak):
    def __init__(self, szybkosc):
        super().__init__("czapla", szybkosc)
    def polow(self):
        print("Tu", self.gatunek, ". Tu łowię ryby")
    def lec(self):
        print("Tu", self.gatunek, ". Nie lecę, będę jadł")


```

ptak › lec()

nie dokonaliśmy żadnej zmiany – nadal jest pole dotyczące gatunku. Natomiast w klasach potomnych dodaliśmy konstruktor. Jest już nam znany jego zapis – **__init__**, zauważmy, że ma on jeden parametr mniej, brakuje gatunku. Został on przeniesiony wiersz niżej i wpisany w formie wywołania. Zapis **super()** pozwala na wywołanie konstruktora klasy, od której dziedziczy dany potomek, więc **super().__init__(„orzel", szybkosc)** oznacza wywołanie dla obiektu klasy **orzel** konstruktora klasy **ptak** ze zdefiniowaną w klasie **orzel** nazwą gatunku podaną jako parametr.

```
ptak_o = orzel(33)
ptak_c = czapla(25)

ptak_o.lec()
ptak_o.polowanie()
ptak_c.lec()
ptak_c.polow()
```

B

```
C:\Python\Projekt1\venv\Scripts\python.exe C:/Pyt
Tu orzel . Startuje, osiągam maksymalnie 33
Tu orzel . Rozpocząłem polowanie
Tu czapla . Nie lecę, będę jadł
Tu czapla . Tu łowię ryby
```

Skutkiem naszych działań jest uproszczenie sposobu tworzenia obiektu poszczególnej klasy B.

## Klasy i metody abstrakcyjne

Teraz na potrzeby naszego przykładu w klasie **ptak** utworzymy nową metodę o nazwie **jakiOdglos**. Będzie to pusta metoda, która powinna być dziedziczona przez klasy potomne.

```
        print("Tu", self.gatunek
              ". Startuje, osiąg

    def jakiOdglos(self):
        pass

class orzel(ptak):
```

Pomimo wywołania tej metody dla obiektów poszczególnych klas nic się nie wydarzy, gdyż utworzona metoda jest po prostu pusta.

```
ptak_o.jakiOdglos()
ptak_c.jakiOdglos()
```

```
C:\Python\Projekt1\venv\Scripts\python.exe
Tu orzel . Startuje, osiągam maksymalnie 33
Tu orzel . Rozpocząłem polowanie
Tu czapla . Nie lecę, będę jadł
Tu czapla . Tu łowię ryby
```

Naszym kolejnym zadaniem jest wymuszenie nadpisania tej metody przez każdą klasę dziedziczącą w taki sposób, aby każdy z gatunków miał własny odgłos.

**1** Powyższa metoda powinna być tak zwaną metodą abstrakcyjną, dla nas oznacza to tylko tyle, że każda klasa dziedzicząca jest zmuszona do nadpisania tej metody. Dodatkowo klasa zawierająca taką metodę nie powinna być nigdy wykorzystywana sama do tworzenia obiektów. Musimy więc zmienić naszą klasę **ptak** na klasę abstrakcyjną i to samo uczynić z metodą **jakiOdglos**. Dzięki zmianom w naszym przykładzie nie będzie można utworzyć obiektu klasy **ptak**, a klasy potomne będą zmuszone do nadpisania pustej metody.

**2** W celu wprowadzania opisanych zmian dodajemy zapis **from abc import ABC, abstractmethod**. Jest to wbudowany moduł w Pythona, który umożliwia proces tworzenia abstrakcyjnych klas oraz metod. Importujemy z niego tylko opisane możliwości. **ABC** służy do tworzenia abstrakcyjnych klas, a **abstractmethod** do tworzenia abstrakcyjnych metod.

```
from abc import ABC, abstractmethod
```

**3** Teraz dodajemy dziedziczenie klasy **ABC** do klasy **ptak**.

```
from abc import ABC, ab

class ptak(ABC):
    def __init__(self,
        self.gatunek =
```

**4** W celu zapisania metody jako abstrakcyjnej musimy skorzystać z tak zwanego **dekoratora**, który poprzedza taką metodę **A**. Po takim zapisie metoda staje się abstrakcyjna i klasy dziedziczące są zmuszone do jej nadpisania.

```
                ". Startuje, osi

A   @abstractmethod
    def jakiOdglos(self):
        pass

class orzel(ptak):
    def __init__(self, szybkos
```

**5** Musimy teraz nadpisać metodę **jakiOdglos** w każdym z potomków, inaczej wywołanie metody nie zadziała poprawnie.

```
class orzel(ptak):
    def __init__(self, szybkosc):
        super().__init__("orzel", szybkosc)
    def polowanie(self):
        print("Tu", self.gatunek, ". Rozpocząłem polowanie")
    def jakiOdglos(self):
        print("argh")

class czapla(ptak):
    def __init__(self, szybkosc):
        super().__init__("czapla", szybkosc)
    def polow(self):
        print("Tu", self.gatunek, ". Tu łowię ryby")
    def lec(self):
        print("Tu", self.gatunek, ". Nie lecę, będę jadł")
    def jakiOdglos(self):
        print("brrbrr")
```

**6** Teraz po wywołaniu metody **jakiOdglos** każda klasa, mimo że korzysta z abstrakcyjnej metody zaimplementowanej w klasie **ptak**, ma możliwość indywidualnego zarządzania daną metodą poprzez wymuszone nadpisanie. W rzeczywistości takie rozwiązanie jest bardzo praktyczne, a wymuszone nadpisanie sprawia, że nie zapomnimy nigdy o dodaniu do kolejnej klasy nadpisania metody, gdyż IDE samo zgłosi nam nasz błąd.

```
ptak_o = orzel(33)
ptak_c = czapla(25)

ptak_o.lec()
ptak_o.polowanie()
ptak_c.lec()
ptak_c.polow()
ptak_o.jakiOdglos()
ptak_c.jakiOdglos()
```

```
C:\Python\Projekt1\venv\Scripts\python.exe C:/Python/P
Tu orzel . Startuje, osiągam maksymalnie 33
Tu orzel . Rozpocząłem polowanie
Tu czapla . Nie lecę, będę jadł
Tu czapla . Tu łowię ryby
```

Do tej pory poznaliśmy już klasy, dowiedzieliśmy się, czym jest dziedziczenie, jak nadpisywać metody, rozszerzać konstruktor, wywoływać konstruktor klasy nadrzędnej, a nawet tworzyć klasy i metody abstrakcyjne. Pozostaje nam jeszcze omówić tematykę związaną z metodami specjalnymi oraz hermetyzacją (enkapsulacją).

## Hermetyzacja

Hermetyzacja, zwana inaczej enkapsulacją, pomimo że brzmi dość egzotycznie, jest tematem łatwiejszym do zrozumienia niż dziedziczenie. Proces hermetyzacji polega w dużym skrócie na ukrywaniu wybranych pól oraz metod klasy w taki sposób, aby nie były dostępne z zewnątrz. Podczas tworzenia

programu jest to dość ważne, gdyż możemy potrzebować:

- udostępnić na zewnątrz klasy tylko te elementy, które są niezbędne,
- uniemożliwić błędną modyfikację klasy,
- ułatwić korzystanie z klasy.

Utworzymy na potrzeby przykładu z hermetyzacją zupełnie nową prostą klasę, dzięki której poznamy zalety takiego tworzenia elementów wewnętrznych klasy, aby pozostawały ukryte.

**1** Tworzymy klasę **auto**, będzie ona zawierała w sobie zupełnie podstawową logikę niezbędną do jazdy autem. Zaczynamy od definicji klasy i jej konstruktora **A**.

**2** Wewnątrz tej klasy mamy konstruktor z trzema polami, zdefiniowanymi na stałe przy tworzeniu obiektu. Dodatkowo mamy kilka metod, które pozwalają na odpalanie auta, zmianę biegów, przyspieszanie i hamowanie. Każde z pól zostało specjalnie zapisane z dwoma znakami podkreślenia. Dzięki temu jest ukryte do modyfikacji. Teraz wywołamy kilka metod

```
auto1 = Auto()
auto1.odpal()
auto1.biegGora()
auto1.przyspiesz()
auto1.hamuj()
auto1.zgas()
```

```
C:\Python\Projekt1\venv\Scripts\python.exe C:/Python/
1
10
0

Process finished with exit code 0
```

```
class Auto:
    def __init__(self):
        self.__odpal = False
        self.__bieg = 0
        self.__szybkosc = 0

    def odpal(self):
        self.__odpal = True
    def zgas(self):
        self.__odpal = False
    def biegGora(self):
        if self.__bieg <= 6: self.__bieg += 1; print(self.__bieg)
    def biegDol(self):
        if self.__bieg >= 0: self.__bieg -= 1; print(self.__bieg)
    def przyspiesz(self):
        if self.__odpal == True and self.__bieg > 0:
            self.__szybkosc += 10; print(self.__szybkosc)
    def hamuj(self):
        if self.__szybkosc >= 10:
            self.__szybkosc -= 10
        else:
            self.__szybkosc = 0
        print(self.__szybkosc)
```

**A**

```
30  auto1.hamuj()
31  auto1.zgas()
32  print(auto1.__bieg)
```

```
C:\Python\Projekt1\venv\Scripts\python.exe C:/Python/Projekt1/main.py
1
10
0
Traceback (most recent call last):
  File "C:\Python\Projekt1\main.py", line 32, in <module>
    print(auto1.__bieg)
AttributeError: 'Auto' object has no attribute '__bieg'

Process finished with exit code 1
```

klasy po utworzeniu obiektu na jej rzecz i sprawdzimy, jak się prezentują dane.

**3** Jak widać, wszystko działa poprawnie, natomiast próba odwołania się do atrybutów naszej klasy jest uniemożliwiona, gdyż przy wywołaniu konkretnego atrybutu pojawi się błąd informujący, że dany atrybut nie istnieje.
Implementacja hermetyzacji w naszej klasie **auto** jest bardzo istotna. Jeśli damy łatwy dostęp do naszej klasy i każdy będzie mógł zmieniać podstawowe pola, może dojść do zepsucia klasy. Ponieważ wartość biegu może być w zakresie od **0** do **6**, modyfikacja pozwalająca na ustawienie biegu na **-2** lub **8** spowoduje brak możliwości działania wbudowanych przez nas metod. Enkapsulacja danych umożliwia więc uniemożliwienie popsucia klasy poprzez ukrycie pól dla normalnej pracy. Minimalizujemy w ten sposób użycie klasy niezgodne z naszym zamysłem.

## WARTO WIEDZIEĆ!

Jak już wcześniej pisaliśmy, w Pythonie nie ma tak naprawdę pól prywatnych lub chronionych – są one jedynie ukryte i dostęp może być utrudniony w zależności od implementacji. Na potrzeby programistów, którzy potrzebują szybkiego środowiska deweloperskiego, rozwiązanie z ukrywaniem jest wystarczające. Tak więc w przypadku pola **__bieg** możemy się do niego odwołać, wpisując **_Auto__bieg**.

```
30  auto1.hamuj()
31  auto1.zgas()
32  print(auto1._Auto__bieg)
```

Możemy zmienić również wartość tego pola i po „zepsuciu” tej wartości metody naszej klasy już nie będą działać prawidłowo.

```
23          print(self.__szybkosc)
24
25
26  auto1 = Auto()
27  print(auto1._Auto__bieg)
28  auto1._Auto__bieg = 8
29  print(auto1._Auto__bieg)
30  auto1.biegGora()
```

W tym wypadku metoda **biegGora** nie została zrealizowana – a co gorsza nie jesteśmy informowani o braku wykonania tej instrukcji, gdyż z założenia wartość pola **bieg** nigdy nie powinna być inna niż **0** przy tworzeniu obiektu i wbudowanymi metodami nie da się wyjść poza bezpieczny zakres.

# 7 Python i komunikacja z internetem

**Python swoją popularność zdobył dzięki bardzo dużej ilości gotowych modułów, które pozwalają na realizowanie skomplikowanych zadań w chwilę. W tym rozdziale zajmiemy się jednym z najciekawszych obszarów, w których Python świetnie się sprawdza – komunikacją z internetem**

## Przygotowania do stworzenia web scrapera

Web scraper to program, który pozwala na wyszukanie tekstu lub innego obiektu na stronie internetowej, a następnie pobranie go, przetworzenie i zapisanie. Jest to bardzo ciekawe zagadnienie, które umożliwia nam pobranie danych statystycznych z różnych stron lub innych ważnych informacji. Zanim jednak będziemy mogli sami stworzyć taki program, musimy zagłębić się nieco w język HTML. Wystarczy, że poznamy podstawy, a będziemy mogli śmiało scrapować strony internetowe.

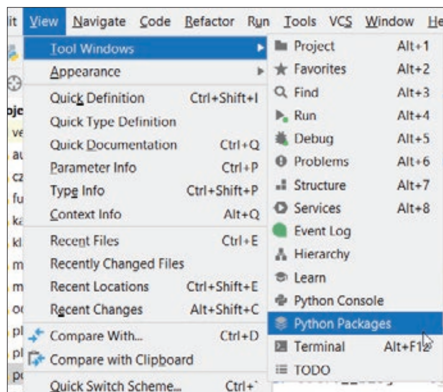
### Instalujemy dodatkowe moduły w PyCharm

1 Pierwszym krokiem w naszych przygotowaniach będzie instalacja dodatkowych pakietów. Jednym z nich jest pakiet **requests**. Instalujemy go, korzystając z pip. Jeśli jednak korzystamy z **PyCharm** i mamy już aktywny projekt, najlepiej zrobić to z poziomu tego IDE, dodając moduł tylko do jednego konkretnego projektu.

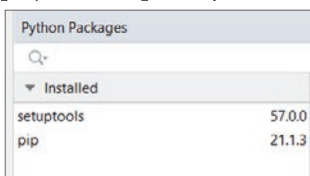



2 Po uruchomieniu PyCharm klikamy w dolnym oknie na zakładkę **Python Packages** A.

**3** Jeśli jej nie ma, klikamy na górnym pasku na **View**, **Tool Windows**, **Python Packages**.

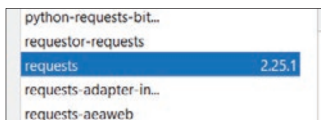



**4** Po lewej stronie znajdziemy wszystkie moduły aktualnie zainstalowane dla naszego projektu oraz pole wyszukiwania.

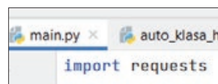



**5** Następnie wpisujemy w pole wyszukiwania **requests** – jest to biblioteka HTTP, dzięki której będziemy mogli pobierać dane ze stron internetowych.

**6** W celu jej zainstalowania klikamy w prawym rogu okna opisu na **Install**.



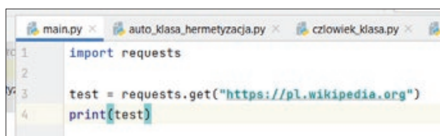

**7** W opisie biblioteki znajdziemy przykładowe wykorzystanie, cechy oraz funkcje i link do dokumentacji. Po zakończeniu instalowania przy danym module pojawi się numer wersji, co oznacza poprawną instalację.



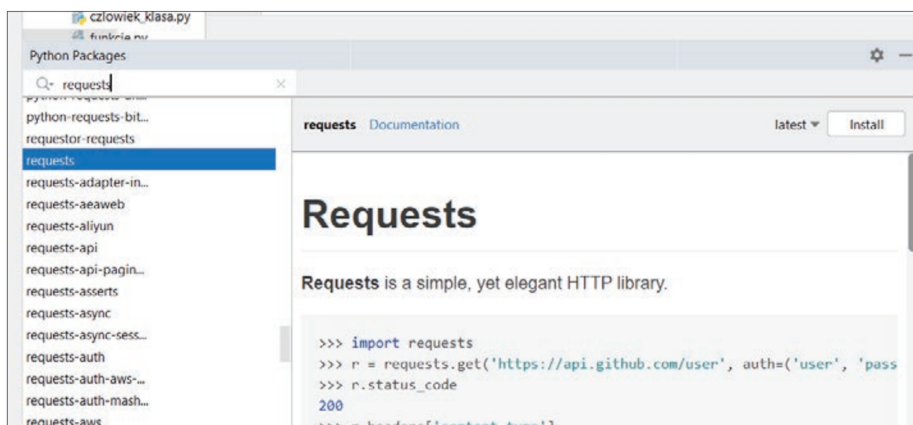

## Łączymy się ze stroną internetową

Dzięki zainstalowanej bibliotece **requests** możemy połączyć się szybko ze stroną internetową.

**1** Tworzymy nowy skrypt i importujemy moduł **requests**.




**2** Następnie tworzymy nowy obiekt i wywołujemy dla niego z modułu **requests** funkcję **get()**.



**3** Otrzymamy w wyniku tego działania kod odpowiedzi strony. W naszym przypadku pojawił się kod 200. Oznacza to, że bez problemów uzyskujemy dostęp do strony. Kod 404 informuje, że strony nie odnaleziono, a kod 403 – o braku praw dostępu do strony. Pełną listę kodów można znaleźć w internecie, ale te trzy na początek nam wystarczą.

```
pole_kola ×   main ×
C:\Python\Projekt1\venv\Scripts\pytho
<Response [200]>

Process finished with exit code 0
```

**4** Poprzez skorzystanie z instrukcji **print** uzyskaliśmy w nawiasach wartość samego obiektu. Jak już wiemy, każdy obiekt ma pewne atrybuty, do których teraz się odwołamy, aby uzyskać lepiej sformatowaną odpowiedź.

**5** Poprzez atrybut **test.status_code** B uzyskamy sam kod odpowiedzi.

```
3   test = requests.get("https:/
4   print(test.status_code)   B

pole_kola ×   main ×
C:\Python\Projekt1\venv\Scripts\pyth
200

Process finished with exit code 0
```

**6** Natomiast w atrybucie **text** umieszczony jest tekst strony, jednak zanim się nim zajmiemy, musimy poznać podstawy HTML, aby zwrócony tekst miał dla nas sens. Domyślnie w atrybucie **text** będzie zapisany praktycznie cały kod źródłowy strony w formatowaniu HTML C.

## Korzystamy z dodatkowych atrybutów

Oczywiście atrybutów jest znacznie więcej. Możemy na przykład pobierać szczegółowe dane dotyczące nagłówka strony.

**1** Podajemy do naszego zapytania zapisanego w obiekcie **test** atrybut **headers**

```
File Edit View Navigate Code Refactor Run Tools VCS Window Help   Projekt1 - main.py
Projekt1 > main.py
main.py ×   auto_klasa_hermetyzacja.py ×   czlowiek_klasa.py ×   klasa_ptaki_dziedzicze
1   import requests
2
3   test = requests.get("https://pl.wikipedia.org")
4   print(test.text)

Run:   pole_kola ×   main ×
C:\Python\Projekt1\venv\Scripts\python.exe C:/Python/Projekt1/main.py
<!DOCTYPE html>
<html class="client-nojs" lang="pl" dir="ltr">   C
<head>
<meta charset="UTF-8"/>
<title>Wikipedia, wolna encyklopedia</title>
<script>document.documentElement.className="client-js";RLCONF={"wgBreakFr
"sysop"],"wgRestrictionMove":["sysop"],"wgIsMainPage":!0,"wgFlaggedRevsPa
"ready","ext.globalCssJs.user":"ready","user":"ready","user.options":"loa
"ext.eventLogging","ext.wikimediaEvents","ext.navigationTiming","ext.uls.
<script>(RLQ=window.RLQ||[]).push(function(){mw.loader.implement("user.op
});});</script>
```

```
3  test = requests.get("https://pl.wikipedia.org")
4  print(test.headers)  D

pole_kola ×   main ×
C:\Python\Projekt1\venv\Scripts\python.exe C:/Python/Projekt1/main.py
{'Date': 'Sun, 27 Jun 2021 22:21:52 GMT', 'Server': 'mw1354.eqiad.wmnet'

Process finished with exit code 0
```

**D**. Pozwala on na wyświetlenie danych dotyczących nagłówka strony.

**2** Możemy również jeszcze bardziej doprecyzować, o jakie dane nam chodzi, przekazując wewnętrzny argument do atrybutu **headers**.

```
3  test = requests.get("https://pl.wiki
4  print(test.headers['Date'])

pole_kola ×   main ×
C:\Python\Projekt1\venv\Scripts\python.exe C
Sun, 27 Jun 2021 22:21:56 GMT

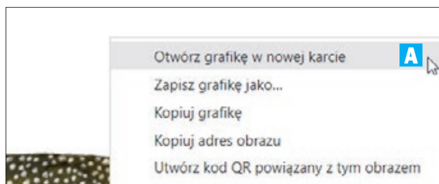
Process finished with exit code 0
```

**3** Uzyskamy w ten sposób dobrze sformatowany łańcuch znaków, a nie listę łańcuchów.

## Pobieramy i zapisujemy obraz, korzystając z modułu requests

Mamy już podstawową wiedzę o łączeniu się ze stronami i pobieraniu danych na ich temat. Teraz połączymy ją z wiadomościami dotyczącymi obsługi plików i pobierzemy obraz ze strony, a następnie zapiszemy go jako plik.

**1** Przechodzimy na stronę, która zawiera obraz, następnie otwieramy tę grafikę w nowym oknie **A**.




**2** Dzięki temu w pasku adresowym będzie dostępny dokładny adres tego konkretnego obrazu **B**.

**3** Zaczynamy od utworzenia obiektu, który przechowa zapytanie **get** **C** naszej strony zawierającej obraz.

**4** Następnie skorzystamy z ciekawej instrukcji **with**, która pozwala na strumieniowanie danych z innych obiektów w locie, w tym wypadku poprzez instrukcję **open**

```
import requests

test = requests.get("https://upload.wikimedia.org/"   C
                    "wikipedia/commons/thumb/a/a4/"
                    "Chilesasaurus.png/1920px-Chilesasaurus.png")
```

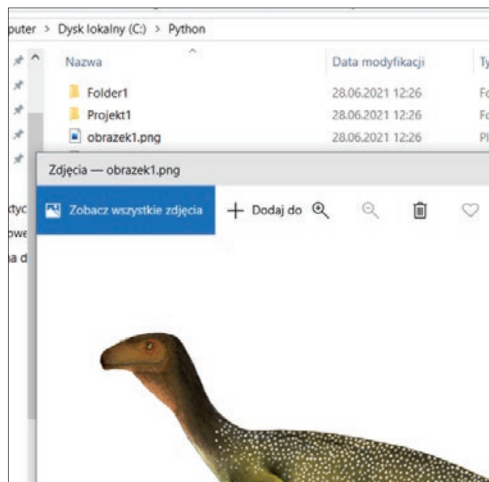
z podaną ścieżką, w której zostanie utworzony nowy plik w trybie do zapisu binarnego. Wewnątrz tego bloku podajemy instrukcję do zapisu całości atrybutu **content**, czyli

```
with open(r'C:\Python\obrazek1.png','wb') as plik:
    plik.write(test.content)
    plik.close()
```



zawartości naszego zapytania, a na koniec zamykamy plik, aby zapisać zmiany.

**5** Po chwili na naszym dysku w określonej lokalizacji pojawi się nowo utworzony plik.




## Przekazujemy argument w zapytaniu get

W celu sprawdzenia, jak działa przekazywanie parametrów, skorzystamy ze strony httpbin.org, która udostępnia możliwość testowania strony WWW. Czasem zachodzi potrzeba przekazania w zapytaniu różnego rodzaju parametrów – nazwy użytkownika, hasła, nazwy do wyszukania itp. Możemy to zrealizować za pomocą polecenia **get**.

**1** Po załadowaniu strony **https://httpbin.org** przechodzimy do podstrony **https://httpbin.org/get** A, gdzie będzie prezentowane nasze zapytanie.

**2** Pozwoli nam to dokładnie sprawdzić, co się dzieje, gdy wysyłamy zapytanie poprzez skrypt Pythona do strony i jak wygląda takie zapytanie w kodzie HTML.

**3** Zaczynamy ponownie od importu modułu **requests**, następnie tworzymy obiekt, którego zadaniem będzie przechowywanie listy parametrów, które chcemy przekazać w zapytaniu.

```
import requests

argumenty = {'mleko':2,'jajka':4}
```

**4** Następnie wykonujemy zapytanie **get**, podając odpowiedni adres strony oraz, po przecinku, jako **params** – nasz obiekt ze słownikiem B.

**5** Po wykonaniu skryptu pojawi się zapytanie. W górnej części będziemy mogli zaobserwować parametry C.

```
test = requests.get('https://httpbin.org/get',params=argumenty)
                                                  B
print(test.text)
```

httpbin.org/get A

```
{
  "args": {},
  "headers": {
    "Accept":
"text/html,application/xhtml+xml,application/xml;q=0.9,image/
tion/signed-exchange;v=b3;q=0.9",
    "Accept-Encoding": "gzip, deflate, br",
    "Accept-Language": "pl-PL,pl;q=0.9,en-US;q=0.8,en;q=0.7",
    "Host": "httpbin.org",
    "Sec-Ch-Ua": "\" Not;A Brand\";v=\"99\", \"Google Chrome\
    "Sec-Ch-Ua-Mobile": "?0",
    "Sec-Fetch-Dest": "document",
    "Sec-Fetch-Mode": "navigate",
    "Sec-Fetch-Site": "none",
    "Sec-Fetch-User": "?1",
    "Upgrade-Insecure-Requests": "1",
    "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64)
Chrome/91.0.4472.114 Safari/537.36",
    "X-Amzn-Trace-Id": "Root=1-60d9a61d-49c7e55b401b3e8d3d90b
  },
  "origin": "31.45.       ",
  "url": "https://httpbin.org/get"
}
```

```
C:\Python\Projekt1\venv\Scripts\python.exe C:/Python/Projekt1/main.py
{
  "args": {
    "jajka": "4",
    "mleko": "2"
  },
  "headers": {
    "Accept": "*/*",
    "Accept-Encoding": "gzip, deflate",
    "Host": "httpbin.org",
    "User-Agent": "python-requests/2.25.1",
    "X-Amzn-Trace-Id": "Root=1-60d9a8c0-38cd06ee35ba5bd8049e5822"
  },
  "origin": "31.45.      ",
  "url": "https://httpbin.org/get?mleko=2&jajka=4"
}
```

C

**6** Jeśli chcemy uzyskać jedynie końcowy URL naszego zapytania **D**, możemy odnieść się do atrybutu utworzonego na potrzeby zapytania **get** obiektu.
Warto zwrócić uwagę, że nasz słownik argumentów został przetworzony, znak **:** został zmieniony na **=**. Dane po lewej stronie to klucze, a po prawej – ich wartości.

```
print(test.url)
```

```
C:\Python\Projekt1\venv\Scripts\python.exe C:/
https://httpbin.org/get?mleko=2&jajka=4

Process finished with exit code 0
```

D

## Korzystamy z funkcji POST i JSON

Do tej pory zbieraliśmy dane z witryn, korzystając z funkcji **GET**, teraz skorzystamy z drugiej najpopularniejszej funkcji modułu **requests**, czyli **POST**. Służy ona do wysyłania danych do serwera, aktualizowania go, dodawania nowych danych. Ponownie jak w poprzednim przykładzie musimy utworzyć słownik na nasze argumenty, które zostaną przekazane.
Słownik wygląda bardzo podobnie. W obiekcie **test** używamy funkcji post **A** oraz zmieniamy adres strony. Następnie wykonujemy skrypt. Odpowiedź, którą otrzymaliśmy, jest zapisana w formacie **JSON**. Możemy

```
import requests

argumenty = {'username':'Janek','password':'12345'}
test = requests.post('https://httpbin.org/post',params=argumenty)
print(test.text)
```

A

```
C:\Python\Projekt1\venv\Scripts\python.exe C:/Python/Projekt1/main.py
{
  "args": {
    "password": "12345",
    "username": "Janek"
  },
  "data": "",
  "files": {},
  "form": {},
  "headers": {
    "Accept": "*/*",
    "Accept-Encoding": "gzip, deflate",
    "Content-Length": "0",
    "Host": "httpbin.org",
    "User-Agent": "python-requests/2.25.1",
    "X-Amzn-Trace-Id": "Root=1-60d9abb0-644b642e13eac73c7f1750ba"
```

```
argumenty = {'username':'Janek','password':'12345'}
test = requests.post('https://httpbin.org/post',params=argumenty)
print(test.json()) B
```

```
C:\Python\Projekt1\venv\Scripts\python.exe C:/Python/Projekt1/main.py
{'args': {'password': '12345', 'username': 'Janek'}, 'data': '', 'files': {}, 'form': {},
'headers': {'Accept': '*/*', 'Accept-Encoding': 'gzip, deflate', 'Content-Length': '0',
'Host': 'httpbin.org', 'User-Agent': 'python-requests/2.25.1', 'X-Amzn-Trace-Id':
'Root=1-60d9ad79-0d17e90a423de26a5e6c7594'}, 'json': None, 'origin': '31.45.:      ',
'url': 'https://httpbin.org/post?username=Janek&password=12345'}

Process finished with exit code 0
```

wykorzystać funkcję **json()** B do utworzenia słownika z odpowiedzi otrzymanej ze strony. Idąc dalej, możemy utworzyć osobny słownik do dalszego przetwarzania, korzystając z tej funkcji, a następnie wyłuskać dane, które nas szczególnie interesują – na przykład hasło i użytkownika C.

Uzyskaliśmy dostęp do tych danych dzięki temu, że wiemy, iż znajdują się w kluczu **args**. Proces, który przed chwilą wykonaliśmy, jest w bardzo dużym uproszczeniu sposobem, w jaki atakujący mogą wykraść dane użytkowników. Wystarczy, że przechwycą nasze zapytanie **POST** wysyłane przez sieć, a następnie, korzystając z kilku przekształceń, uzyskają w kilka sekund wszystkie argumenty, w tym hasła i loginy wysyłane wewnątrz takiego zapytania. Oczywiście problem ten dotyczy zapytań typu HTTP, a nie zabezpieczonych typu HTTPS, które nie pozwalają na przesyłanie tego typu danych w otwartym tekście.

## Moduł webbrowser

Warto wiedzieć, że Python ma również moduł do obsługi przeglądarki internetowej, do której możemy przekazywać parametry.

```
import webbrowser

webbrowser.open("https://komputerswiat.pl")   A
```

```
C:\Python\Projekt1\venv\Scripts\python.exe C:/Python/Projekt1/mai

Process finished with exit code 0
```

Komputer Świat - Komputery, Te
komputerswiat.pl

Jeśli chcemy otworzyć jakąś stronę w przeglądarce, wystarczy zaimportować moduł **webbrowser** A, a następnie wywołać me-

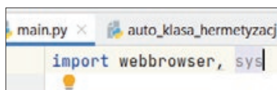
```
test = requests.post('https://httpbin.org/post',params=argumenty)
slownik = test.json()
print(slownik['args'])
```

```
C:\Python\Projekt1\venv\Scripts\python.exe C:/Python/Projekt1/main.py
{'password': '12345', 'username': 'Janek'} C

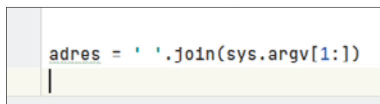
Process finished with exit code 0
```

todę **open**, podając jako argument pełen adres strony.
Dla przykładu chcemy, aby nasz skrypt po otrzymaniu adresu otwierał nam stronę **Google Maps** ze znalezionym adresem, wskazanym przez nas jako argument. Do tego będziemy potrzebowali zarówno modułu **webbrowser**, jak i **sys**.

**1** Zaczynamy od importu niezbędnych bibliotek.




**2** Następnie tworzymy zmienną **adres**, która będzie przyjmować wprowadzone przez użytkownika dane w wierszu polecenia. Służy do tego wyrażenie **sys.argv**.



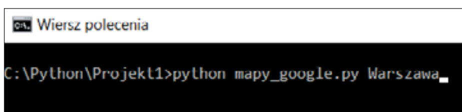

**3** Następnie podajemy dokładną ścieżkę, jaka ma być otworzona w przeglądarce, dodatkowo dołączając naszą zmienną **adres**.

```
webbrowser.open("https://google.com/maps/place/"+adres)
```

**4** Teraz zapisujemy skrypt i nadajemy mu nazwę, na przykład **mapy_google.py**.

Teraz wywołamy nasz skrypt, korzystając z Wiersza polecenia w systemie Windows.

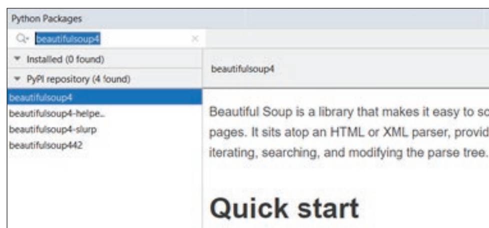
**1** Uruchamiamy Wiersz polecenia i przechodzimy do lokalizacji, w której znajduje się skrypt. Następnie wprowadzamy komendę **python mapy_google.py Warszawa**




**2** Po chwili zostanie otwarta przeglądarka z podanym przez nas adresem **A**.

## Korzystamy z Beautiful Soup

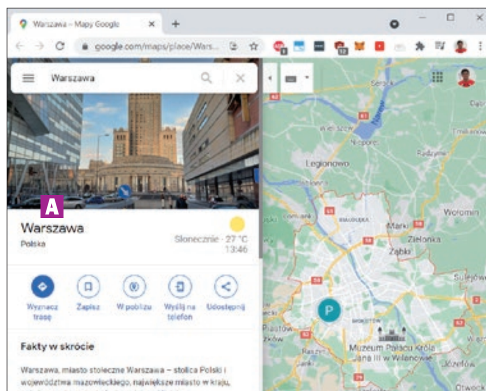
Już w pierwszym przykładzie wykorzystania modułu **requests**, przy pobraniu całej treści strony, mogliśmy zauważyć, że tekst opakowany w znaczniki kodu HTML jest bardzo trudny do odczytania. Dlatego też stworzono parsery, specjalne programy, których zadaniem jest przetworzenie mało czytelnego kodu HTML do formatu, który będziemy mogli wygodnie wykorzystywać. Python ma wbudowany parser, dodatkowo jednak moduł **Beautiful Soup** pomaga wyciągnąć później przetworzone dane w prosty sposób. Nie jest to domyślnie instalowany moduł, więc korzystając z pip lub interfejsu PyCharm instalujemy go dodatkowo – nazwa tego pakietu to **beautifulsoup4**, można również użyć skróconej formy **bs4**.







**1** W celu użycia tego modułu musimy go zaimportować. Dodatkowo importujemy też moduł **requests** **B**.



```
<meta charset="utf-8"/>
<title>Chilesaurus - Wikipedia, wolna encyklopedia</title> D
<script>document.documentElement.className="client-js";RLCONF={"wgBreakFrames":!1,
 "wgSeparatorTransformTable":[",\t.",", \t,"],"wgDigitTransformTable":["",""],
 "wgDefaultDateFormat":"dmy","wgMonthNames":["","styczeń","luty","marzec","kwiecień","maj",
 "czerwiec","lipiec","sierpień","wrzesień","październik","listopad","grudzień"],
 "wgRequestId":"fdfa7016-bf65-4c57-823e-1a87e180a311","wgCSPNonce":!1,
 "wgCanonicalNamespace":"","wgCanonicalSpecialPageName":!1,"wgNamespaceNumber":0,
 "wgPageName":"Chilesaurus","wgTitle":"Chilesaurus","wgCurRevisionId":63824815,
```

```
strona = requests.get("https://pl.wikipedia.org/wiki/Chilesaurus")

dane = bs4.BeautifulSoup(strona.text) C
```

**2** Teraz tworzymy zapytanie **get** dla konkretnej strony, a następnie przypisujemy do nowego obiektu dane, które zostaną przekazane do funkcji **BeautifulSoup** **C**.

**3** Samo wyświetlenie później tego obiektu za wiele nam nie pomoże. Nadal będzie mnóstwo kodu HTML **D**.
Dzięki Beautiful Soup możemy uzyskać informacje na temat ilości poszczególnych elementów w kodzie strony. Dodatkowo do każdego elementu jest przypisany tekst i dodatkowy kod, możemy więc odnosić się do konkretnych elementów strony. Dzięki temu możemy wskazać dokładny wycinek strony, który nas interesuje, i na nim się skoncentrować.

## Korzystamy z Pythona do połączenia dwóch maszyn

Do tego zadania będziemy potrzebować kolejnej biblioteki sieciowej Pythona, tym razem jest to **sockets**. Służy ona do obsługi tak zwanych gniazd i nawiązywania połączeń w sieci oraz sprawdzania dostępności różnego typu portów oraz adresów.

**1** Ten przykład zrealizujemy w IDLE, gdyż udostępnia ono możliwość korzystania z interaktywnej sesji, więc na bieżąco będziemy mogli zweryfikować, czy połączenie

## OSTRZEŻENIE WYBORU PARSERA

```
C:\Python\Projekt1\main.py:5: GuessedAtParserWarning: No parser was explicitly specified,
 so I'm using the best available HTML parser for this system ("html.parser"). This usually
 isn't a problem, but if you run this code on another system, or in a different virtual
 environment, it may use a different parser and behave differently.

The code that caused this warning is on line 5 of the file C:\Python\Projekt1\main.py. To
 get rid of this warning, pass the additional argument 'features="html.parser"' to the
 BeautifulSoup constructor.

  dane = bs4.BeautifulSoup(strona.text)
```

Jeśli pojawi się ostrzeżenie takie jak na zrzucie ekranu powyżej, należy dodać do metody **Beautiful Soup** dodatkowy argument, który definiuje na sztywno wybór parsera. Po modyfikacji problem nie powinien występować.

```
dane = bs4.BeautifulSoup(strona.text, features="html.parser")
```

zostało nawiązane pomyślnie. Warto wiedzieć, że w Pythonie, aby połączyć dwie maszyny, jedna musi pełnić rolę serwera, a druga klienta. W skrócie jedna czeka na połączenie, druga się łączy.

```
IDLE Shell 3.9.5
File Edit Shell Debug Options Window Help
Python 3.9.5 (tags/v3.9.5:0a7dcbd, May  3 2021, 17:27:52) [MSC v
D64)] on win32
Type "help", "copyright", "credits" or "license()" for more info
>>> import socket
>>> sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
>>> port = 50005 B
>>> |
```

**2** Zaczynamy od strony serwera. Importujemy moduł **socket** i definiujemy nasze gniazdo **s**.

```
*IDLE Shell 3.9.5*
File Edit Shell Debug Options Window Help
Python 3.9.5 (tags/v3.9.5:0a7dcbd, May  3 2021, 17:27:52)
D64)] on win32
Type "help", "copyright", "credits" or "license()" for mo
>>> import socket
>>> s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
>>> |
```

**3** Sprawdzamy nasz adres IP, na przykład w Wierszu polecenia wpisując **ipconfig**.

```
Wireless LAN adapter Wi-Fi:

   Connection-specific DNS Suffix  . :
   IPv4 Address. . . . . . . . . . . : 192.168.43.5
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : 192.168.43.1
```

**4** Teraz definiujemy nasz serwer, podając adres IP oraz port **A**, na którym będziemy oczekiwać na połączenie. Następnie uruchamiamy usługę nasłuchu, po czym umożliwiamy dostęp do sieci dla naszego programu.

```
>>> s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
>>> s.bind(('192.168.43.5', 50005)) A
>>> s.listen(5)
>>>
```

Alert funkcji Zabezpieczenia Windows

Zapora Windows Defender zablokowała niektóre funkcje tej aplikacji

Zapora Windows Defender zablokowała niektóre funkcje programu Python we wszystkich sieciach publicznych i prywatnych.

Nazwa: Python

Wydawca: Python Software Foundation

Ścieżka: C:\users\krzys\appdata\local\programs\python\python39\pythonw.exe

Zezwól programowi Python na połączenia w tych sieciach:

☑ Sieci prywatne, takie jak sieci domowe lub firmowe

☑ Sieci publiczne, takie jak w portach lotniczych i kawiarniach (niezalecane, ponieważ takie sieci na ogół mają słabe zabezpieczenia lub nie mają ich wcale).

Jakie ryzyko wiąże się z zezwoleniem aplikacji na dostęp przez zaporę?

Zezwalaj na dostęp | Anuluj

**5** Następnie na innym komputerze w tej samej sieci domowej otwieramy sesję **IDLE** i również importujemy moduł, tworzymy gniazdo i zmienną dla portu naszego serwera **B**.

**6** Teraz nawiązujemy połączenie, korzystając z funkcji **connect**.

```
>>> port = 50005
>>> sock.connect(('192.168.43.5',port))
>>> |
```

```
>>> s.bind(('192.168.43.5', 50005))
>>> s.listen(5)
>>> clientsocket,clientaddress = s.accept() C
>>>
```

**7** Następnie w kodzie naszego serwera dodajemy wiersz umożliwiający przyjmowanie połączeń **C**.

**8** Teraz po wpisaniu w sesji serwera polecenia **clientaddress** pojawi się informacja o podłączonym kliencie. Jak widać, połączenie nadeszło z portu 1055.

```
>>> clientsocket,clientaddress = s.accept()
>>> clientaddress
('192.168.43.5', 1055)
>>> |
```

Po nawiązaniu połączenia w ten sposób możemy wysyłać między maszynami zarówno wiadomości, jak i pliki.

# Tworzymy skanery sieci

Zaczniemy od najprostszych rozwiązań i będziemy sukcesywnie dodawać kolejne funkcjonalności do naszego kodu w celu stworzenia skanera sieciowego, który będzie korzystał ze skanowania adresów IP.
Nie potrzebujemy żadnego dodatkowego modułu do pracy naszego skanera. Wystarczą nam wbudowane narzędzia i biblioteki.

**1** Skorzystamy z najprostszej formy skanowania – użyjemy polecenia **ping** **A**, które jest dostępne w każdym systemie.

```
Skaner_sieci_1.py - D:/KS_Praca/python4s/Skaner_sieci_1.py (3.8.3)
File  Edit  Format  Run  Options  Window  Help
import os

os.system('cls')

print('#'*40)
ip_do_sprawdzenia = input('Ip do sprawdzenia: ')
print('-'*40)
os.system('ping {}'.format(ip_do_sprawdzenia))   A
print('-'*40)
input('Nacisnij dodowlny klawisz, aby wyjsc')
```

**2** Poprzez polecenie **input** sami wprowadzimy konkretny adres IP **B**, który zamierzamy przeskanować. Po zatwierdzeniu klawiszem [enter] rozpocznie się skan.

```
================ RESTART: D:/KS_Praca/pytho
########################################
Ip do sprawdzenia: 192.168.1.100   B
```

**3** Jest to podstawowe zautomatyzowanie skanowania jednego adresu.

```
Pinging 192.168.1.100 with 32 bytes of data:
Reply from 192.168.1.100: bytes=32 time<1ms TTL=64
Reply from 192.168.1.100: bytes=32 time<1ms TTL=64
```

Jeśli zamierzamy napisać automatyczny skaner, który wykryje istniejące urządzenia w naszej podsieci, będziemy musieli jednak napisać bardziej skomplikowany kod.

## Skaner aktywnych urządzeń w całej podsieci

Jest to już znacznie bardziej skomplikowany skrypt, który musi analizować wiele różnych parametrów i w odpowiedni sposób je interpretując, zwracać nam wynik. Nas interesuje jedynie, czy dany host (maszyna) jest aktywny na danym adresie IP, czy też nie. Na tej podstawie możemy wykryć, czy komputer jest aktywny w sieci.
Pełną treść skryptu **A** znajdziemy w pliku **Skaner_sieci_2.py** (w **KŚ+**).
Po załadowaniu go możemy zapoznać się z jego działaniem.

```
Skaner_sieci_2.py - D:/KS_Praca/python4s/Skaner_sieci_2.py (3.8.3)   A
File  Edit  Format  Run  Options  Window  Help
import subprocess   B

C def ping_ip(ip):
    (output, error) = subprocess.Popen(([ 'ping', ip, '-n', '1']), stdin=subprocess.PIPE, stdout=subprocess.PIPE,
    if b'bytes=32' in output:
        return "DZIALA"
    elif b'Destination host unreachable.' in output:
        return "BRAK ODPOWIEDZI"
    elif error:
        return "Blad DNS"
    else:
        return "NIEZNANY"

addr = input("Podaj pierwsze 3 człony adresu IP np. XXX.XXX.XXX. : ")
a = input("Podaj poczatek zakresu skanowania od 1 do 255: ")        D
b = input("Podaj koniec zakresu skanowania od 1 do 255: ")
for ip in range(int(a),int(b)+1):
    ip = str(addr)+str(ip)
    ip = ip.strip('\n')
    response = ping_ip(ip)
    result = ('%s,%s \n' % (ip, response))
    print(result)
```

Na początku importujemy moduł **subprocess** B, który pozwoli nam na uruchomienie podprocesów, dzięki czemu będziemy mogli przeskanować każdy adres IP osobno. Następnie, podając przedrostek **def** C, rozpoczynamy definicję funkcji **ping_ip**, odpowiedzialnej za skanowanie adresu przy każdym wywołaniu. W zależności od wyniku skanowania zwraca ona różne odpowiedzi. Ostatni akapit sprawia, że użytkownik może podać konkretną podsieć oraz zakres adresów do przeskanowania D. Parametry przekazywane są do pętli, która przekazuje adres po adresie do naszej funkcji **ping_ip** i oczekuje na wynik. Dość szybko jesteśmy w stanie przeanalizować nawet całą podsieć E.

```
>>>
================ RESTART: D:/KS_Praca/python4s/Skaner_sieci_2.py
Podaj pierwsze 3 człony adresu IP np. XXX.XXX.XXX. : 192.168.1.
Podaj poczatek zakresu skanowania od 1 do 255: 100
Podaj koniec zakresu skanowania od 1 do 255: 110
192.168.1.100,DZIALA          E

192.168.1.101,DZIALA

192.168.1.102,DZIALA

192.168.1.103,DZIALA

192.168.1.104,DZIALA

192.168.1.105,DZIALA

192.168.1.106,NIEZNANY

192.168.1.107,BRAK ODPOWIEDZI
```

## Skaner otwartych portów

Oczywiście samo to, że dane urządzenie jest dostępne w sieci i ma przydzielony adres IP, może nam nie wystarczać. Dlatego też możemy napisać skaner otwartych portów. Dzięki niemu będziemy wiedzieli, czy dane urządzenie ma otwarte porty, przez co jest ryzyko, że nie jest odpowiednio zabezpieczone. Kod programu można znaleźć w pliku **Skaner_sieci_3.py** (w **KŚ+**). Skrypt służący do sprawdzania otwartych portów jest jeszcze bardziej rozbudowany. Korzysta z wbudowanego modułu **socket**,

### CIEKAWY BANER

Jeśli chcemy, aby nasz program wyświetlał w trakcie swojej pracy taki baner A, możemy skorzystać z modułu **pyfiglet**. Instalujemy go, wpisując w Wierszu polecenia komendę **pip install pyfiglet** B. Potem importujemy moduł **pyfiglet** do naszego programu i wpisujemy treść banera.

```
Wiersz polecenia
<~ Przerwany Skan ~>                                A
C:\Python>python Skaner_sieci_3.py 192.168.193.210
SKANER PORTOW
PYTHON 3
3.210
0:55.555991
efiltrowany ~>
```

```
C:\Windows\system32>pip install pyfiglet   B
Collecting pyfiglet
  Using cached https://files.pythonhosted.org/package
5/pyfiglet-0.8.post1-py2.py3-none-any.whl
Installing collected packages: pyfiglet
Successfully installed pyfiglet-0.8.post1
WARNING: You are using pip version 19.2.3, however ve
You should consider upgrading via the 'python -m pip

C:\Windows\system32>
```

```
*Skaner_sieci_3.py - D:/KS_Praca/python4s/Skaner_sieci_3.py (3.8.3)*
File  Edit  Format  Run  Options  Window  Help

#!/bin/python3
import pyfiglet
import sys
import socket
from datetime import datetime as dt

def nl():
    print('\n')


if len(sys.argv) == 2:
    target = socket.gethostbyname(sys.argv[1]) A
else:
    nl()
    print("Niepoprawnie zdefiniowany parametr wejścia:(")
    print("Syntax --> python3 portscanner.py <IP Address>")
    nl()

#Baner B
baner = pyfiglet.figlet_format("SKANER PORTOW KS SPECIAL 2020")
print(baner)

nl()
print("~" * 50)
print("Skanowany host ~> {}".format(target))
print("Czas startu: {}".format(dt.now()))
print("~" * 50)
nl()

try: C          D
    for port in range (18,81):
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        socket.setdefaulttimeout(1)
        result = s.connect_ex((target, port))

        if result == 0:
            print("<~ Port {} jest otwarty i niefiltrowany ~>".format(port))
        s.close

except KeyboardInterrupt: E
    print("<~ Przerwany Skan ~>")
    sys.exit()
except socket.gaierror:
    print("<~ !Nazwa hosta niepoprawna! ~>")
    sys.exit()
except socket.gaierror:
    print("<~ !Błąd połączenia! ~>")
    sys.exit()
```

który pozwoli nam na weryfikację, czy port jest faktycznie otwarty.
Ten skrypt będzie uruchamiany zupełnie inaczej niż wszystkie znane nam do tej pory. Musimy otworzyć Wiersz polecenia, przejść do lokalizacji ze skryptem, a następnie wpisać polecenie **python Skaner_sieci_3.py [adres_ip_do_skanowania]**.
Po podaniu adresu IP pierwsza instrukcja warunkowa naszego kodu zostanie spełniona, adres zostanie przypisany do zmiennej **target** **A** i będzie wykorzystany w dalszej części programu.
W sekcji **Baner** **B** zajmujemy się generowaniem tekstu informacyjnego dla naszego skryptu.
Natomiast główna logika skryptu jest realizowana w bloku instrukcji **try** **C**, gdzie port po porcie sprawdzamy, czy podane w zakresie **range** **D** porty są otwarte dla danego adresu IP. Zakres portów możemy dowolnie modyfikować. Jeśli dany port jest zamknięty, nie zwracamy żadnej informacji. Jeżeli jest otwarty, wyświetlamy to w oknie programu.
Na końcu dodana została obsługa wyjątków **E**. Można dalej rozbudować skaner portów, łącząc go ze skanerem adresów IP, i wykonywać zaawansowane skany całych podsieci wraz z portami. Możemy zmienić zaszyte w kodzie skryptu wartości portów na podawane przez użytkownika i przekazywane w zmiennych. Możliwości rozbudowy jest wiele.

```
D:\KS_Praca\python4s>
D:\KS_Praca\python4s>python Skaner_sieci_3.py 192.168.1.100
```

# Raspberry Pi

# Jak zrobić czujnik odległości w Pythonie

**Jeśli interesują nas projekty „zrób to sam", warto zainteresować się Raspberry Pi. Jest to bardzo mały komputer z dodatkowymi wyjściami cyfrowymi, które pozwalają na obsługę wielu czujników. Dzięki temu możemy na przykład stworzyć czujnik odległości**



Nowa edycja małego komputera – Raspberry Pi 4 Model B, zachowując dość niską cenę, ma o wiele większą wydajność w stosunku do poprzedniej wersji. Są dostępne trzy różne rodzaje tego modelu, które różnią się ilością pamięci RAM – 1 GB, 2 GB, 4 GB. Najnowsza „Malinka" kosztować nas będzie od 160 do 270 złotych, w zależności od tego, ile pamięci będziemy chcieli mieć w naszym urządzeniu. Za najniższą cenę możemy mieć komputer, który wyposażony jest w czterordzeniowy 64-bitowy procesor Broadcom BCM2711 taktowany na 1,5 GHz, obsługę Wi-Fi 2,4 oraz 5 GHz, Bluetooth 5.0, Gigabit Ethernet, 2x USB 3.0, 2x USB 2.0, 2x micro HDMI (ze wsparciem dla 4K).

Taka specyfikacja pozwala wykorzystać Raspberry Pi 4 na przykład jako stację multimedialną do starszego telewizora. Dodatkowo mamy do dyspozycji 40 pinów GPIO, które pozwalają na podłączanie wielu czujników i płytek stykowych do kontroli przeróżnych mikrokontrolerów.

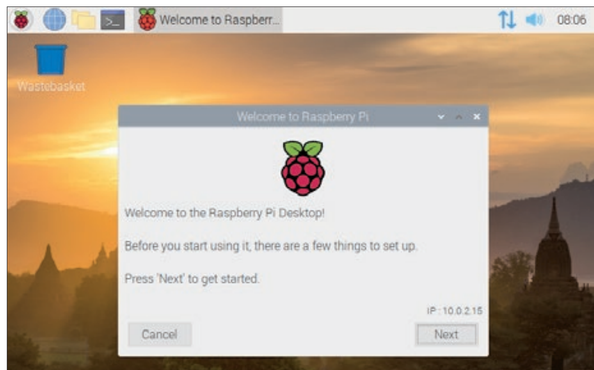
## RASPBERRY PI ZERO

W cenie około 100 złotych możemy kupić znacznie mniejsze urządzenie z minimalnym wyposażeniem, które w głównej mierze komunikuje się przez sieć bezprzewodową. Raspberry Pi Zero W jest o połowę mniejsze niż nowszy model 4 B. Nie ma też tak dobrych podzespołów, nie pozwoli więc na utworzenie stacji multimedialnej. Jeśli natomiast planujemy stworzenie jakiegoś gadżetu w zamkniętej obudowie i jesteśmy ograniczeni wymiarami, jest to świetne rozwiązanie.

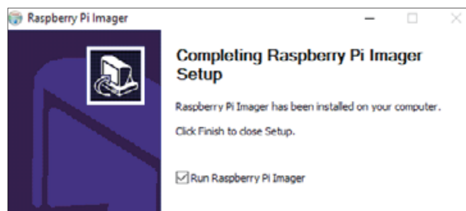


# System operacyjny

Jak każdy komputer, „Malinka" również korzysta z systemu operacyjnego. Polecany przez jej twórców system to **Raspberry Pi OS** (dawniej Raspbian). Jest to dystrybucja systemu Linux oparta na Debianie, która została dopasowana idealnie do podzespołów małego komputera. Dzięki temu nie będziemy mieli problemu z obsługą żadnych złączy i wszystko będzie działać od razu po zainstalowaniu systemu. (Alternatywnie możemy zainstalować inne systemy, na przykład Noobs lub Windows 10 IoT).
Raspberry Pi OS oprócz tego, że jest idealnie przystosowany do małego komputera, to przy instalacji dodatkowo instaluje pakiety edukacyjne do programowania, jak **Python**, **Scratch**, **Sonic Pi**, **Java** i inne. Dzięki temu od razu możemy zacząć tworzenie własnych skryptów, które w połączeniu z różnego rodzaju czujnikami pozwolą na automatyzację codziennych zadań lub wykonywanie konkretnego, potrzebnego nam zadania.
Ze względu na konstrukcję urządzenia Raspberry Pi głównym nośnikiem danych domyślnie jest **karta microSD**. Jeśli chcemy zaoszczędzić, możemy kupić samo urządzenie bez karty w zestawie, a wykorzystać kartę, którą już mamy, i samemu przygotować ją do uruchomienia w miniaturowym komputerze.




**1** Pobieramy z KŚ+ (lub ze strony **www.raspberrypi.org/downloads**) program **Raspberry Pi Imager for Windows**, instalujemy go i uruchamiamy.

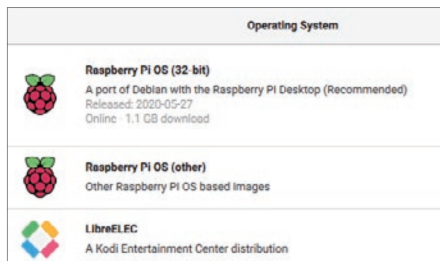



**2** Następnie umieszczamy w naszym komputerze kartę microSD, którą przeznaczyliśmy do „Malinki".

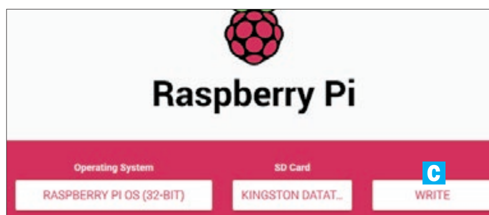
**3** Klikamy na **Choose OS** **A**.

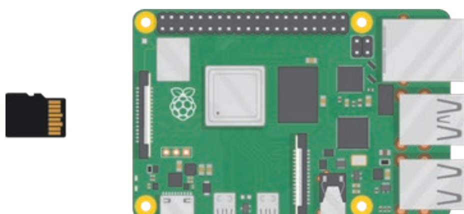



**4** Teraz wybieramy pierwszą pozycję od góry – **Raspberry Pi OS (32-bit)**.




**5** Następnie klikamy na **Choose SD Card** **B** i wskazujemy naszą kartę.



**6** Na koniec klikamy na przycisk **Write** **C**.

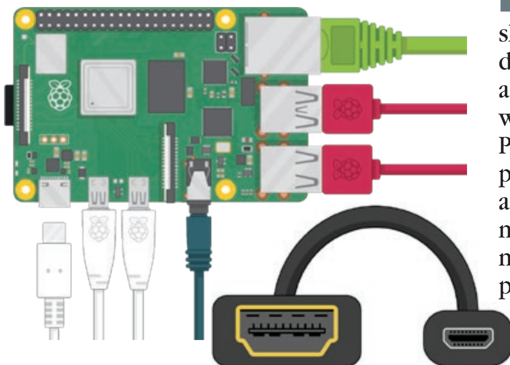
**7** Teraz możemy przełożyć kartę do naszego urządzenia Raspberry PI.

# Dodatkowe peryferia



■ Przed pierwszym uruchomieniem podłączamy **mysz i klawiaturę USB** do naszego Raspberry Pi w celu wykonania wstępnej konfiguracji (później będziemy mogli korzystać z klawiatury i myszy poprzez Bluetooth, nie zajmując portów USB urządzenia).

■ Następnie musimy podłączyć naszą „Malinkę" do **wyświetlacza**. (Nawet jeśli docelowo mamy korzystać z tego komputera jak z serwera, który jedynie przetwarza dane z czujników i udostępnia je w sieci, to bez wyświetlacza nie da się przeprowadzić konfiguracji systemu). Raspberry Pi możemy połączyć zarówno z telewizorem, jak i zwykłym monitorem. Raspberry Pi 4 ma dwa porty micro HDMI i umożliwia podłączenie nawet dwóch ekranów jednocześnie. Musimy jedynie zaopatrzyć się w **kabel HDMI–micro HDMI** lub odpowiedni **adapter**, jeśli chcemy wykorzystać złącze DVI albo VGA w monitorze.



■ Dodatkowo warto zakupić **obudowę** na nasz mikrokomputer, aby uniknąć jego uszkodzenia. Jest bardzo duży wybór nieoficjalnych tańszych zamienników. Jeśli mamy drukarkę 3D, możemy też sami wydrukować sobie obudowę – darmowe projekty można znaleźć w sieci.

■ Do większych modeli Raspberry Pi możemy również podłączyć **urządzenia audio** przez standardowy port 3,5 mm. Jeśli korzystamy z kabla HDMI do podłączenia na przykład telewizora, nie musimy się tym przejmować, gdyż kanały audio zostaną przeniesione i dźwięk będzie odtwarzany bez dodatkowych połączeń przewodowych.

■ **Połączenie z internetem** możemy uzyskać, początkowo korzystając ze standardowego portu Ethernet w Raspberry Pi, a później skonfigurować łączność bezprzewodową przez Wi-Fi. W przypadku modelu Pi Zero, w którym nie ma takiego portu, możemy wykorzystać adapter USB – Ethernet, który pozwoli na dostęp do sieci przewodowej.



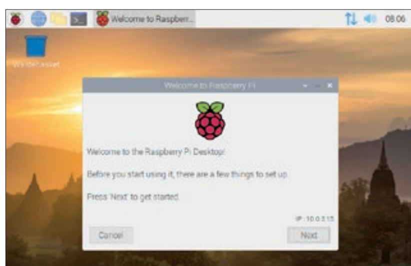
# Pierwsze uruchomienie

**Uwaga!** Raspberry Pi nie ma standardowego przełącznika, który umożliwia włączenie i wyłączenie urządzenia. Od razu po podłączeniu przewodu zasilającego „Malinka” rozpoczyna rozruch (można własnoręcznie zaprojektować włącznik, jest to jednak nieco skomplikowane). Dlatego też, w celu ochrony plików na naszej karcie oraz ogólnie całego urządzenia, należy zawsze zwracać szczególną uwagę na proces uruchamiania.

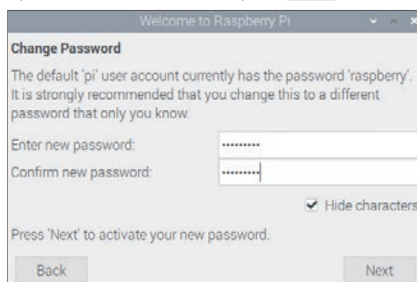
**1** Po podłączeniu zasilania na płytce Raspberry Pi zaświeci się czerwona dioda, która sygnalizuje proces bootowania. Na podłączonym wyświetlaczu w lewym górnym rogu pojawią się cztery maliny.



**2** Potem zobaczymy pulpit systemu Raspberry Pi OS z ekranem powitalnym. Rozpoczynamy konfigurację, klikając na **Next**. Wybieramy kraj **Poland**, język **Polish** i strefę czasową **Warsaw**. Klikamy na **Next**.



**3** **Uwaga!** Domyślny użytkownik systemu to pi, a hasło to **raspberry** – zalecana jest jednak zmiana, gdyż poprzez otwarte porty i uruchomione różnego rodzaju usługi osoby trzecie mogłyby przejąć kontrolę nad naszym urządzeniem na późniejszym etapie użytkowania. Dlatego też tworzymy nowe hasło dla użytkownika **pi** i klikamy na **Next**.



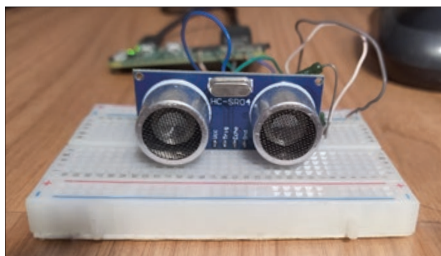

**5** W kolejnym kroku zezwalamy na aktualizację, klikając na **Next**. Jeśli chcemy skorzystać z połączenia bezprzewodowego i nie jesteśmy podłączeni do sieci, pojawi się okno wyszukiwania sieci bezprzewodowych.




**6** Po zakończeniu aktualizacji należy ponownie uruchomić system i można zacząć z niego korzystać.

# Programujemy czujnik odległości

Zanim zaczniemy zabawę z czujnikami czy ledami, musimy mieć chociaż podstawową wiedzę z zakresu elektrotechniki – w sposób bezpośredni będziemy pracować z prądem. Należy zachować szczególną ostrożność.



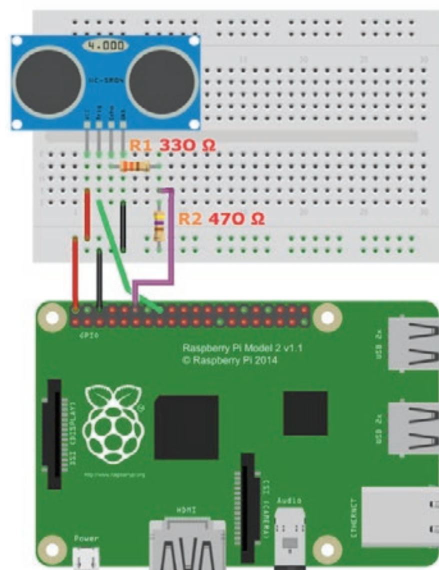
Do zbudowania podstawowego czujnika, który pozwoli nam mierzyć odległość, można wykorzystać tani ultradźwiękowy czujnik **HC-SR04** (kosztuje około 8 złotych). Będziemy musieli wykonać kilka połączeń na płytce stykowej i zaprogramować działanie czujnika w języku Python.

## Elementy układanki

A oto, co dokładnie będzie nam potrzebne do wykonania kolejnych kroków:

- urządzenie z grupy Raspberry Pi (dowolny model),
- czujnik ultradźwiękowy **HC-SR04**,
- płytka stykowa,
- oporniki 330 omów oraz 470 omów (lub ich zamienniki utworzone z innych oporników w układach równoważnych),
- kilka przewodów miedzianych.

Całość podłączamy w sposób przedstawiony na schemacie.




## Programowanie w Pythonie

Teraz, aby nasze połączenie zadziałało, musimy zainstalować bibliotekę **GPIO** dla Pythona, która pozwoli nam na kontrolę wyjść i wejść naszego urządzenia Raspberry Pi.

**1** Uruchamiamy terminal i wykonujemy krok po kroku kolejne komendy, zatwierdzając je klawiszem [enter] i, jeśli zajdzie taka potrzeba, autoryzując całość podaniem hasła.

```
sudo apt-get install git-core
sudo apt-get update
sudo apt-get upgrade

git clone git://git.drogon.net/wiringPi

cd wiringPi
git pull origin

cd wiringPi
./build
```

**2** Sprawdzamy, czy instalacja przebiegła poprawnie i nasz interfejs GPIO działa – po wpisaniu i zatwierdzeniu komendy **gpio readall** powinniśmy zobaczyć taki widok.

```
pi@raspberrypi:~ $ gpio readall
 +-----+-----+---------+------+---+-Pi ZeroW-+---+------+---------
 | BCM | wPi |   Name  | Mode | V | Physical | V | Mode | Name
 +-----+-----+---------+------+---+----++----+---+------+---------
 |     |     |    3.3v |      |   |  1 || 2  |   |      | 5v
 |   2 |   8 |   SDA.1 |   IN | 1 |  3 || 4  |   |      | 5v
 |   3 |   9 |   SCL.1 |   IN | 1 |  5 || 6  |   |      | 0v
 |   4 |   7 | GPIO. 7 |   IN | 0 |  7 || 8  | 0 | IN   | TxD
 |     |     |      0v |      |   |  9 || 10 | 1 | IN   | RxD
 |  17 |   0 | GPIO. 0 |   IN | 0 | 11 || 12 | 0 | IN   | GPIO.
 |  27 |   2 | GPIO. 2 |   IN | 0 | 13 || 14 |   |      | 0v
 |  22 |   3 | GPIO. 3 |   IN | 0 | 15 || 16 | 0 | IN   | GPIO.
 |     |     |    3.3v |      |   | 17 || 18 | 0 | IN   | GPIO.
 |  10 |  12 |    MOSI |   IN | 0 | 19 || 20 |   |      | 0v
 |   9 |  13 |    MISO |   IN | 0 | 21 || 22 | 0 | IN   | GPIO.
 |  11 |  14 |    SCLK |   IN | 0 | 23 || 24 | 1 | IN   | CE0
```

**3** Teraz możemy przejść do pisania skryptu w języku Python, który pozwoli na pomiar odległości. Gotowy skrypt znajduje się w pliku **odle1x.py** (w **KŚ+**). Są w nim komentarze wyjaśniające jego działanie. A oto podstawowe informacje:

**A** Na początku skryptu importujemy potrzebne biblioteki, następnie, korzystając z instrukcji zawartych w bibliotece **GPIO**, ustawiamy tryb pracy urządzenia oraz piny wykorzystywane do sterowania czujnikiem. Przypisujemy interfejsy wejścia i wyjścia dla wybranych pinów.

**B** Następnie w części **def distance** wykonujemy pomiar odległości, bazując na tym, że prędkość dźwięku to **34 300 cm/s**. Czujnik ultradźwiękowy wysyła dźwięk do przeszkody, a następnie wychwytuje go

**A**

```
#Biblioteki
import RPi.GPIO as GPIO
import time
#GPIO Mode (BOARD / BCM)
GPIO.setmode(GPIO.BCM)
#Ustawiamy piny GPIO
GPIO_TRIGGER = 18
GPIO_ECHO = 24
#Ustawiamy kierunek GPIO (IN / OUT)
GPIO.setup(GPIO_TRIGGER, GPIO.OUT)
GPIO.setup(GPIO_ECHO, GPIO.IN)
```

**B**

```
def distance():
    # Ustawiamy Trigger na wysoki
    GPIO.output(GPIO_TRIGGER, True)
    # Ustawiamy Trigger po 0.01ms na niski
    time.sleep(0.00001)
    GPIO.output(GPIO_TRIGGER, False)
    StartTime = time.time()
    StopTime = time.time()
    # zachowujemy StartTime
    while GPIO.input(GPIO_ECHO) == 0:
        StartTime = time.time()
    # zachowujemy time of arrival
    while GPIO.input(GPIO_ECHO) == 1:
        StopTime = time.time()
    # roznica czasu
    TimeElapsed = StopTime - StartTime
    # mnozymy przez predkosc dzwieku (34300 cm/s)
    # dzielimy przez 2 - droga do celu i do czujnika
    distance = (TimeElapsed * 34300) / 2
    return distance
```

**C**

```
if __name__ == '__main__':
    try:
        while True:
            dist = distance()
            print ("Zmierzona odleglosc = %.1f cm" % dist)
            time.sleep(1)
        # Reset po wciśnięciu CTRL + C
    except KeyboardInterrupt:
        print("Pomiar zatrzymany przez uzytkownika")
        GPIO.cleanup()
```

w drodze powrotnej, dlatego wykonujemy dzielenie przez 2.

**C** W ostatniej części programu umieszczona została nieskończona pętla, która co sekundę wywołuje naszą funkcję do pomiaru odległości. Program można zatrzymać, wciskając kombinację klawiszy ctrl+C.

**4** Po utworzeniu skryptu na naszym urządzeniu należy go uruchomić, wpisując polecenie **python odle1x.py** w tej samej lokalizacji, w której jest zapisany skrypt.

## Co dalej

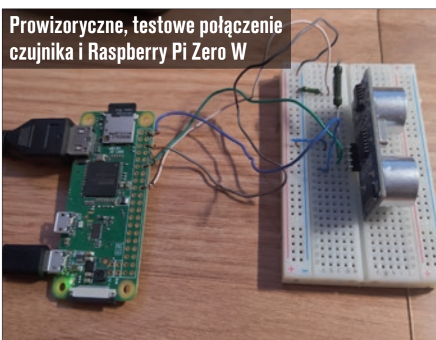
Nasz skrypt działa – ale najlepiej potraktować to jako początek zabawy. Warto go rozbudować, a cały układ zamontować w obudowie.

Przykładowe wykorzystanie czujnika w praktyce: umieszczamy go w odpowiedniej odległości od drzwi i za każdym razem, gdy będą otwierane, a czujnik wykryje znaczną zmianę mierzonej odległości, zostanie wysłany do nas e-mail lub wykonane zdjęcie modułem kamery, który również można podłączyć do Raspberry Pi.

Liczba czujników i możliwości konfiguracyjnych jest nieograniczona. Można wykonywać nawet różne instalacje automatyzujące, które będą sprawdzać się równie dobrze jak drogie produkty smart home.

```
pi@raspberrypi:~ $ python odle1x.py
Zmierzona odleglosc = 32.3 cm
Zmierzona odleglosc = 32.2 cm
Zmierzona odleglosc = 32.8 cm
Zmierzona odleglosc = 32.5 cm
Zmierzona odleglosc = 32.7 cm
Zmierzona odleglosc = 32.4 cm
Zmierzona odleglosc = 28.7 cm
Zmierzona odleglosc = 40.7 cm
Zmierzona odleglosc = 44.1 cm
Zmierzona odleglosc = 43.7 cm
Zmierzona odleglosc = 43.5 cm
Zmierzona odleglosc = 45.2 cm
Zmierzona odleglosc = 43.7 cm
Zmierzona odleglosc = 52.7 cm
Zmierzona odleglosc = 139.5 cm
Zmierzona odleglosc = 67.8 cm
Zmierzona odleglosc = 62.2 cm
Zmierzona odleglosc = 61.2 cm
^CPomiar zatrzymany przez uzytkownika
pi@raspberrypi:~ $
```

Działający skrypt – ultradźwiękowy czujnik odległości



Prowizoryczne, testowe połączenie czujnika i Raspberry Pi Zero W

# JAK SKORZYSTAĆ Z E-WYDANIA KSIĄŻKI

W KŚ+ znajdziemy **e-wydanie tej Biblioteczki, obraz ISO dołączonej do niej płyty z plikami szkoleniowymi i programami oraz plik PDF książki do pobrania.**

**1** Otwieramy stronę **ksplus.pl**. Logujemy się **A** (używamy konta z serwisu **Komputerswiat.pl**). Jeżeli nie mamy konta, klikamy na **B**, by się zarejestrować.

B Załóż konto Logowanie A
Zarejestruj kod

**2** Po zalogowaniu się możemy zarejestrować kod nadrukowany na płycie dołączonej do książki. Wystarczy kliknąć na **C** i przepisać kod.

Moje konto
C Zarejestruj kod

**3** Uzyskamy w ten sposób dostęp do e-wydania **D** i do bonusowego obrazu płyty **E**. Do serwisu KŚ+ możemy logować się z dowolnego urządzenia z dostępem do internetu.

CZYTAJ E-WYDANIE D
PROGRAMY E
BONUSY

**UWAGA! W KŚ+ ZA DARMO E-WYDANIE KSIĄŻKI ORAZ PLIK ISO PŁYTY**



**Krzysztof Dziedzic**
*autor książki, informatyk*

# TO, CO TRZEBA WIEDZIEĆ O PYTHONIE

Python jest najpopularniejszym językiem programowania ostatnich lat. Za jego pomocą są tworzone aplikacje sieciowe, jest wykorzystywany w skomplikowanych obliczeniach, a nawet – w grach. Jest stosowany przez takie firmy, jak Facebook, Google, Dropbox czy Netflix.

Ważną cechą Pythona jest to, że pozwala rozbudowywać gotowe programy, także napisane w innych językach, na przykład w Javie lub C++ – można do nich dodawać pojedyncze moduły tworzone właśnie w Pythonie i rozszerzające ich możliwości.

Ta książka zawiera najważniejsze informacje, zarówno podstawowe, jak i bardziej zaawansowane, pozwalające zacząć programować w Pythonie. Poznamy typy zmiennych, operatory, instrukcje warunkowe i pętle. Nauczymy się definiować funkcje i pisać pierwsze programy. Zobaczymy w praktyce, na czym polega programowanie obiektowe i obsługa błędów.

Na DVD i w serwisie KŚ+ (ksplus.pl) znajdziemy skrypty do przedstawionych w książce wskazówek oraz narzędzia do programowania w Pythonie.

**CENA 16,90 ZŁ**
W TYM 5% VAT




Nr 4/2021 (114)



KOMPUTER ŚWIAT BIBLIOTECZKA